



VS Pascal

SC26-4320-1

Language Reference

Release 2





VS Pascal

SC26-4320-1

Language Reference

Release 2

Second Edition (December 1988)

This edition replaces and makes obsolete the previous edition, SC26-4320-0.

This edition applies to Release 2 of VS Pascal, Program Number 5668-767 (Compiler and Library) and 5668-717 (Library only) and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

The changes for this edition are summarized under "Summary of Changes" in Appendix A. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's program may be used. Any functionally equivalent program may be used instead.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publishing, P. O. Box 49023, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. © Copyright International Business Machines Corporation 1987, 1988. All rights reserved.

Preface

This book provides definition of the VS Pascal programming language and its syntax. No information other than that contained in this book should be considered definition of the VS Pascal programming language and syntax.

When used in conjunction with *VS Pascal Application Programming Guide*, this book can help you in writing, debugging, and maintaining VS Pascal applications.

This manual is not a learner's guide. It is to be used by people who are involved in the task of creating or maintaining applications, and who are already familiar with Pascal and with programming in high-level languages. Some helpful Pascal textbooks are listed in the "Bibliography" on page 280.

How This Manual Is Organized

If you are unfamiliar with VS Pascal you may want to read the first three introductory chapters. If you are an experienced VS Pascal user you can turn directly to the chapter that interests you.

In general, reference chapters begin with a "quick-reference" chart that presents keywords and topics in functional groupings. Explanations of individual keywords and topics then follow in alphabetic order.

Chapter 1, "How to Read Syntax Diagrams" on page 2, explains how to read VS Pascal syntax diagrams.

Chapter 2, "VS Pascal Program Elements" on page 8, introduces some basic elements of VS Pascal programs.

Chapter 3, "Structure of VS Pascal Programs" on page 18, describes the two types of compilable units (the program unit and the segment unit) and how VS Pascal programs are structured.

Chapter 4, "Declarations" on page 24, explains VS Pascal declarations in alphabetic order.

Chapter 5, "Constants" on page 36, explains VS Pascal constants.

Chapter 6, "Data Types" on page 44, provides a chart of VS Pascal data types in functional order. Explanations of each data type follow in alphabetic order.

Chapter 7, "Variables" on page 92, discusses some general principles of VS Pascal variables and explains how different types of variables are referenced.

Chapter 8, "Routines" on page 102, discusses some general principles of VS Pascal routines and provides, in alphabetic order, explanations of the VS Pascal predefined routines. In addition, explanations are provided for some non-predefined VS Pascal library routines.

Chapter 9, "Expressions" on page 194, defines what constitutes an expression in VS Pascal and explains the various types of expressions.

Chapter 10, "Statements" on page 208, provides a chart of VS Pascal statements. Explanations of each statement follow in alphabetic order.

Chapter 11, “Compiler Directives” on page 230, provides a chart of VS Pascal compiler directives in functional order. Explanations of each directive follow in alphabetic order.

Appendix A, “Summary of Changes” on page 242, summarizes the additions and enhancements VS Pascal Release 2 makes to VS Pascal Release 1.

Appendix B, “Predefined Identifiers” on page 246, provides an all-inclusive reference chart of VS Pascal predefined identifiers: constants, data types, routines, and variables.

Appendix C, “Options for Opening Files” on page 252, provides a brief description of the options used for opening files.

Appendix D, “Syntax Diagrams” on page 254, groups together, in alphabetic order, all of the syntax diagrams used in this manual.

Appendix E, “Migration Considerations” on page 270, summarizes points to consider when migrating from VS Pascal Release 1 to Release 2, as well as migration considerations from Pascal/VS Release 2.2 to VS Pascal Release 1.

“Glossary” on page 277, defines some important terms used in this manual.

“Bibliography” on page 280, lists the IBM VS Pascal publications, as well as related text books that provide information on the Pascal programming language.

Industry Standards

The VS Pascal Compiler and Library, Release 2, supports the specifications of the American National Standard Pascal Computer Programming Language (ANSI/IEEE770X3.97-1983). This standard is referred to in this manual as the ANSI Standard, or simply Standard Pascal.

The VS Pascal Compiler and Library, Release 2, also supports the specifications of the International Standards Organization Programming Language - Pascal (ISO 7185-1983) at Level 0. This standard is referred to in this manual as the ISO standard.

The VS Pascal Compiler and Library, Release 2, also supports the Federal Information Processing Standard Publication (FIPS PUB) 109.

Typographical Considerations

VS Pascal Language Reference helps you recognize important terms and features in three ways:

IBM Extensions To Standard Pascal Are Printed In Blue

This helps you distinguish Standard Pascal keywords and features from the keywords and features that IBM has added to make VS Pascal more powerful and easier to use.

Keywords Appear in Capitals

To help you recognize keywords that must be coded exactly as presented, the following appear in all capital letters:

- Pascal reserved words
- Predefined identifiers
- User-defined identifiers.

Keywords used in a generic sense appear in lowercase. For example, the reserved word TYPE appears in capitals, but general references to data type are always in lowercase.

Double-Byte and Single-Byte Characters Are Clearly Represented

Throughout *VS Pascal Language Reference*, double-byte character set (DBCS) characters and some single-byte character set (SBCS) characters are represented as follows:

- Each dot/letter combination, as in ".A", represents one DBCS character.
- The "<" and the ">" represent the shift-out and shift-in characters, respectively.
- "b" represents one SBCS blank.
- ".b" represents one DBCS blank.

Contents

Chapter 1. How to Read Syntax Diagrams	2
No Parameters	2
Required Parameters	2
Optional Parameters	3
Multiple Parameters	4
Default Parameters	5
Chapter 2. VS Pascal Program Elements	8
Identifiers	8
Reserved Words	9
Special Symbols	10
Comments	11
Double-Byte Character Set (DBCS) Comments	12
Literals	12
Hexadecimal and Binary Literals	14
Double-Byte Character Set (DBCS) Literals	15
Double-Byte Character Set (DBCS) Mixed Literals	15
Chapter 3. Structure of VS Pascal Programs	18
Program Units	18
Segment Units	20
Linking Units to Form a Program	22
Chapter 4. Declarations	24
Lexical Scope of Identifiers	24
Declaration Order	27
VS Pascal Declarations	27
CONST Declaration	27
DEF Declaration	28
LABEL Declaration	28
REF Declaration	29
STATIC Declaration	30
TYPE Declaration	30
VALUE Declaration	31
VAR Declaration	31
Chapter 5. Constants	36
Types of Constants	36
Predefined Constants	37
Structured Constants	37
Chapter 6. Data Types	44
The Basic Data Types	44
Simple Data Types	44
Pointer Data Type	44
String Pointer Data Type	45
Structured Data Types	45
Creating Your Own Data Types	45
Type Compatibility	46
Implicit Type Conversion	46
Same Data Types	47
Compatible Data Types	47

Assignment Compatibility	48
Storage, Packing, and Alignment of Variables	49
VS Pascal Data Types	49
ALFA Data Type	50
ALPHA Data Type	50
ARRAY Data Type	51
BOOLEAN Data Type	53
CHAR Data Type	55
DBCS Fixed String Data Type	56
Enumerated Scalar Data Type	57
FILE Data Type	59
GCHAR Data Type	60
GSTRING Data Type	61
INTEGER Data Type	64
Pointer Data Type	66
REAL Data Type	67
RECORD Data Type	69
SBCS Fixed String Data Type	76
SET Data Type	77
SHORTREAL Data Type	79
SPACE Data Type	81
STRING Data Type	81
STRINGPTR Data Type	84
Subrange Data Type	86
TEXT Data Type	88
Chapter 7. Variables	92
Predefined Variables	93
Subscripted Variables	93
Array Variables	93
STRING Variables	94
GSTRING Variables	95
Error Checking	95
Field Referencing	95
Pointer Referencing	96
File Referencing	97
Space Referencing	98
Chapter 8. Routines	102
Routine Declarations	102
Routine Parameters	105
Pass-by-Value Parameters	105
Pass-by-VAR Parameters	105
Pass-by-CONST Parameters	106
Formal Routine Parameters	106
Routines That Can Be Passed as Parameters	107
Function Results	107
Routine Directives	108
EXTERNAL Routine Directive	109
FORTRAN Routine Directive	109
FORWARD Routine Directive	110
GENERIC Routine Directive	110
MAIN Routine Directive	112
REENTRANT Routine Directive	112
Predefined Routines	113
ABS Function	118

ADDR Function	118
ARCTAN Function	118
CHR Function	119
CLOCK Function	119
CLOSE Procedure	119
COLS Function	119
COMPRESS Function	120
COS Function	121
DATETIME Procedure	121
DELETE Function	122
DISPOSE Procedure	123
DISPOSEHEAP Procedure	123
EOF Function	124
EOLN Function	125
EXP Function	125
FLOAT Function	126
GET Procedure	126
GSTR Function	126
GTOSTR Function	127
HALT Procedure	128
HBOUND Function	128
HIGHEST Function	129
INDEX Function	130
LBOUND Function	130
LENGTH Function	131
LN Function	132
LOWEST Function	132
LPAD Procedure	133
LTOKEN Procedure	134
LTRIM Function	135
MARK Procedure	136
MAX Function	137
MAXLENGTH Function	137
MCOMPRESS Function	138
MDELETE Function	139
MIN Function	139
MINDEX Function	140
MLENGTH Function	140
MLTRIM Function	141
MRINDEX Function	141
MSUBSTR Function	142
MTRIM Function	143
NEW Procedure	143
NEWHEAP Procedure	146
ODD Function	149
ORD Function	150
PACK Procedure	150
PAGE Procedure	151
PARMS Function	151
PDSIN Procedure	151
PDSOUT Procedure	152
PRED Function	152
PUT Procedure	153
QUERYHEAP Procedure	153
RANDOM Function	154
READ Procedure (for Record Files)	154

READ and READLN Procedures (for Text Files)	155
READSTR Procedure	160
RELEASE Procedure	162
RESET Procedure	163
RETCODE Procedure	163
REWRITE Procedure	163
RINDEX Function	164
ROUND Function	165
RPAD Procedure	165
SEEK Procedure	166
SIN Function	167
SIZEOF Function	167
SQR Function	167
SQRT Function	168
STOGSTR Function	168
STR Function	169
SUBSTR Function	170
SUCC Function	171
TERMIN Procedure	171
TERMOUT Procedure	171
TOKEN Procedure	172
TRACE Procedure	173
TRIM Function	174
TRUNC Function	174
UNPACK Procedure	175
UPDATE Procedure	176
USEHEAP Procedure	177
WRITE Procedure (for Record Files)	177
WRITE and WRITELN Procedures (for Text Files)	178
WRITESTR Procedure	184
Additional Routines	186
CMS Procedure	186
ITOHs Function	187
LPAD Procedure	187
ONERROR Procedure	188
PICTURE Function	189
RPAD Procedure	191
Chapter 9. Expressions	194
Operators	197
The NOT Operators	197
The Multiplication Operators	197
The Addition Operators	198
The Relational Operators	199
BOOLEAN Expressions	199
Constant Expressions	201
Logical Expressions	202
Function Calls	203
Ordinal Conversions	203
Set Constructors	204
Chapter 10. Statements	208
VS Pascal Statements	209
ASSERT Statement	209
Assignment Statement	209
CASE Statement	211

Compound Statement	214
CONTINUE Statement	215
Empty Statement	215
FOR Statement	216
GOTO Statement	219
IF Statement	220
LEAVE Statement	222
Procedure Call	223
REPEAT Statement	223
RETURN Statement	224
WHILE Statement	225
WITH Statement	225
Chapter 11. Compiler Directives	230
VS Pascal Compiler Directives	231
%CHECK Directive	231
%CPAGE Directive	232
%ENDSELECT Directive	233
%INCLUDE Directive	233
%LIST Directive	234
%MARGINS Directive	235
%PAGE Directive	235
%PRINT Directive	235
%SELECT Directive	236
%SKIP Directive	236
%SPACE Directive	236
%TITLE Directive	237
%UHEADER Directive	237
%WHEN Directive	238
%WRITE Directive	240
Appendix A. Summary of Changes	242
Appendix B. Predefined Identifiers	246
Appendix C. Options for Opening Files	252
Appendix D. Syntax Diagrams	254
Appendix E. Migration Considerations	270
From VS Pascal Release 1 to VS Pascal Release 2	270
From Pascal/VS Release 2.2 to VS Pascal Release 1	273
Glossary	277
Bibliography	280
Index	281

Figures

1.	Syntax of a VS Pascal Identifier	8
2.	Examples of Valid and Invalid Identifiers	9
3.	VS Pascal Reserved Words	9
4.	Summary of VS Pascal Special Symbols	10
5.	Symbols That Are Reserved Words	11
6.	Example of a Nested Comment	12
7.	Example of DBCS Comments	12
8.	Syntax of Literals	13
9.	Examples of Literals	14
10.	Example of a DBCS Literal	15
11.	Example of a DBCS Mixed Literal	16
12.	Syntax of Valid Mixed Strings	16
13.	Syntax of Canonical Mixed Strings	16
14.	Syntax of a Program Unit	18
15.	Structure of a Program Unit	19
16.	Example of a Program Unit	19
17.	Syntax of a Segment Unit	21
18.	Structure of a Segment Unit	21
19.	Example of a Segment Unit	21
20.	Linking a Program Unit with Segment Units	22
21.	Summary of VS Pascal Declarations	24
22.	Scope of Identifiers	25
23.	Nesting Structure of a Program	26
24.	Syntax of the CONST Declaration	27
25.	Syntax of the DEF Declaration	28
26.	Syntax of the LABEL Declaration	29
27.	Syntax of the REF Declaration	29
28.	Syntax of the STATIC Declaration	30
29.	Syntax of the TYPE Declaration	31
30.	Syntax of the VALUE Declaration	31
31.	Syntax of the VAR Declaration	32
32.	Examples of VS Pascal Declarations	33
33.	Categories of Constants	36
34.	Syntax of Constants	36
35.	Syntax of Structured Constants	38
36.	Example of an Array Constant	39
37.	Example of a Record Constant	39
38.	Examples of Structured Constants with Variant Record Fields	40
39.	Example of a Combination of an Array and Record Constant	41
40.	Syntax of a Data Type	45
41.	Implicit Type Conversions Performed by VS Pascal	46
42.	Examples of Type Compatibility	48
43.	Summary of VS Pascal Data Types	49
44.	Operators and Predefined Functions for Type ALFA	50
45.	Operators and Predefined Functions for Type ALPHA	51
46.	Syntax of the ARRAY Data Type	51
47.	Examples of ARRAY Declarations	52
48.	Predefined Routines for Type ARRAY	52
49.	Operators and Predefined Functions for Type BOOLEAN	53
50.	Relational Operators on Type BOOLEAN	54
51.	Operators and Predefined Functions for Type CHAR	55
52.	Operators and Routines for the DBCS Fixed String Data Type	56

53.	Syntax of the Enumerated Scalar Data Type	57
54.	Examples of Enumerated Scalar Data Types	58
55.	Predefined Functions for Enumerated Scalar Data Type	58
56.	Syntax of the FILE Data Type	59
57.	Examples of FILE Declarations	59
58.	Routines for Record File Variables	60
59.	Operators and Predefined Functions for Type GCHAR	61
60.	Syntax of the GSTRING Data Type	62
61.	Operators and Predefined Routines for Type GSTRING	62
62.	How to Apply Binary Operators to DBCS Characters, DBCS Fixed Strings, and DBCS Strings	63
63.	How to Convert DBCS Strings on Assignment	64
64.	Operators and Predefined Functions for Type INTEGER	64
65.	Syntax of the Pointer Data Type	66
66.	Example of Pointer Declarations	66
67.	Operators and Predefined Routines for the Pointer Data Type	67
68.	Operators and Predefined Functions for Type REAL	68
69.	Syntax of the RECORD Data Type	69
70.	Examples of Simple RECORD Declarations	70
71.	Example of a Record Declaration with a Tag Field	72
72.	Storage of a Record with a Tag Field	72
73.	Example of a Record Declaration with a Back Reference Tag Field	73
74.	Storage of a Record with a Back Reference Tag Field	73
75.	Example of a Record Variant with No Tag Field	74
76.	Storage of a Record Variant with No Tag Field	74
77.	Example of a Record with Offset Qualified Fields	75
78.	Example of an Offset Qualifier on a Tag Field	76
79.	Predefined Functions for Type RECORD	76
80.	Operators and Routines for the SBCS Fixed String Data Type	77
81.	Syntax of the SET Data Type	77
82.	Example of SET Declarations	78
83.	Operators and Functions for Type SET	79
84.	Operators and Predefined Functions for Type SHORTREAL	80
85.	Syntax of the SPACE Data Type	81
86.	Functions for the SPACE Data Type	81
87.	Syntax of the STRING Data Type	81
88.	Operators and Routines for Strings (Byte-Oriented)	82
89.	Functions for Strings (Character-Oriented)	83
90.	How to Apply Binary Operators to SBCS Characters, SBCS Fixed Strings, and SBCS Strings	84
91.	How to Convert Strings on Assignment	84
92.	Example of the Type STRINGPTR	85
93.	Operators and Predefined Routines for the STRINGPTR Data Type	85
94.	Syntax of the Subrange Data Type	86
95.	Examples of Subrange Scalars	87
96.	Examples of Subranges with the Same Base Type	87
97.	Predefined Functions for Type Subrange Scalar	88
98.	Procedures and Functions for Type TEXT	88
99.	Syntax of a Variable Reference	92
100.	Example of Variables Used in Their Entirety	92
101.	Example of Array Indexing	94
102.	Examples of Valid and Invalid Subscripting for a STRING Data Type	94
103.	Example of GSTRING Indexing	95
104.	Example of Field Referencing	96
105.	Example of Pointer Referencing	97
106.	Example of File Referencing	98

107. Examples of Space Referencing	99
108. Example of Invalid Space Referencing	99
109. Syntax of Routine Declarations	103
110. Examples of Routine Declarations	104
111. Example of Conformant String Parameters	106
112. Predefined Routines That Can Be Passed as Parameters	107
113. Routines That Can Be Passed as Parameters	107
114. Example of a Recursive Function	108
115. Example of a Function Returning a Record	108
116. Example of the EXTERNAL Directive	109
117. Example of the GENERIC Routine Directive	111
118. Example of Coding Before Development of the GENERIC Routine Directive	111
119. Summary of Conversion Routines	113
120. Summary of Data Inquiry Routines	114
121. Summary of Data Movement Routines	114
122. Summary of General Routines	114
123. Summary of Input/Output Routines	114
124. Summary of Mathematical Routines	115
125. Summary of Storage Management Routines	116
126. Summary of String Routines for SBCS and DBCS Strings	116
127. Summary of String Routines for Mixed Strings	117
128. Summary of System Access Routines	117
129. Definition of the ABS Function	118
130. Definition of the ADDR Function	118
131. Definition of the ARCTAN Function	118
132. Definition of the CHR Function	119
133. Definition of the CLOCK Function	119
134. Definition of the CLOSE Procedure	119
135. Definition of the COLS Function	120
136. Definition of the COMPRESS Function	120
137. Examples of the COMPRESS Function	120
138. Definition of the COS Function	121
139. Definition of the DATETIME Procedure	121
140. Example of the Date and Time Format	121
141. Definition of the DELETE Function	122
142. Example of the DELETE Function	122
143. Definition of the DISPOSE Procedure	123
144. Definition of the DISPOSEHEAP Procedure	123
145. Definition of the EOF Function	124
146. Example of Testing for End-of-File Condition	124
147. Definition of the EOLN Function	125
148. Example of Copying a Text File	125
149. Definition of the EXP Function	125
150. Definition of the FLOAT Function	126
151. Definition of the GET Procedure	126
152. Definition of the GSTR Function	127
153. Example of the GSTR Function	127
154. Definition of the GTOSTR Function	127
155. Example of the GTOSTR Function	128
156. Definition of the HALT Procedure	128
157. Definition of the HBOUND Function	128
158. Example of the HBOUND Function	129
159. Definition of the HIGHEST Function	129
160. Example of the HIGHEST Function	129
161. Definition of the INDEX Function	130

162.	Example of the INDEX Function	130
163.	Definition of the LBOUND Function	131
164.	Example of the LBOUND Function	131
165.	Definition of the LENGTH Function	131
166.	Examples of the LENGTH Function	132
167.	Definition of the LN Function	132
168.	Definition of the LOWEST Function	132
169.	Example of the LOWEST Function	133
170.	Definition of the LPAD Procedure	133
171.	Examples of the LPAD Procedure	134
172.	Definition of the LTOKEN Procedure	134
173.	Example of the LTOKEN Procedure	135
174.	Definition of the LTRIM Function	135
175.	Examples of the LTRIM Function	136
176.	Definition of the MARK Procedure	136
177.	Example of Using MARK and RELEASE within a Single Heap	137
178.	Definition of the MAX Function	137
179.	Definition of the MAXLENGTH Function	138
180.	Example of the MAXLENGTH Function	138
181.	Definition of the MCOMPRESS Function	138
182.	Example of the MCOMPRESS Function	138
183.	Definition of the MDELETE Function	139
184.	Example of the MDELETE Function	139
185.	Definition of the MIN Function	140
186.	Definition of the MINDEX Function	140
187.	Example of the MINDEX Function	140
188.	Definition of the MLENGTH Function	140
189.	Example of the MLENGTH Function	141
190.	Definition of the MLTRIM Function	141
191.	Example of the MLTRIM Function	141
192.	Definition of the MRINDEX Function	142
193.	Example of the MRINDEX Function	142
194.	Definition of the MSUBSTR Function	142
195.	Example of the MSUBSTR Function	143
196.	Definition of the MTRIM Function	143
197.	Example of the MTRIM Function	143
198.	Definition of the NEW Procedure	144
199.	Example of the NEW Procedure (Form 1)	144
200.	Example of the NEW Procedure (Form 2)	145
201.	Example of the NEW Procedure (Form 3)	146
202.	Definition of the NEWHEAP Procedure	147
203.	Example of the NEWHEAP Procedure	148
204.	Definition of the ODD Function	149
205.	Definition of the ORD Function	150
206.	Definition of the PACK Procedure	150
207.	Example of the PACK Procedure	151
208.	Definition of the PAGE Procedure	151
209.	Definition of the PARMS Function	151
210.	Definition of the PDSIN Procedure	152
211.	Definition of the PDSOUT Procedure	152
212.	Definition of the PRED Function	152
213.	Example of the PRED Function	153
214.	Definition of the PUT Procedure	153
215.	Definition of the QUERYHEAP Procedure	153
216.	Definition of the RANDOM Function	154
217.	Definition of the READ Procedure (for Record Files)	154

218. Example of the READ Procedure for Record Files	154
219. Example of Multiple Variables on READ	155
220. Definition of the READ and READLN Procedures (for Text Files)	155
221. Example of the READLN Procedure	156
222. Example of Multiple Variables on READ and READLN	157
223. Example of the READLN Procedure with Lengths	158
224. Example of Reading GCHAR and GSTRING Data	159
225. Example of Reading Mixed String Data	160
226. Definition of the READSTR Procedure	161
227. Example of the READSTR Procedure	161
228. Example of Code Equivalents to READSTR	162
229. Definition of the RELEASE Procedure	162
230. Definition of the RESET Procedure	163
231. Definition of the RETCODE Procedure	163
232. Definition of the REWRITE Procedure	164
233. Definition of the RINDEX Function	164
234. Examples of the RINDEX Function	164
235. Definition of the ROUND Function	165
236. Example of the ROUND Function	165
237. Definition of the RPAD Procedure	166
238. Example of the RPAD Procedure	166
239. Definition of the SEEK Procedure	166
240. Definition of the SIN Function	167
241. Definition of the SIZEOF Function	167
242. Definition of the SQR Function	168
243. Definition of the SQRT Function	168
244. Definition of the STR Function	168
245. Example of the STOGSTR Function	169
246. Definition of the STR Function	169
247. Example of the STR Function	169
248. Definition of the SUBSTR Function	170
249. Example of the SUBSTR Function	170
250. Definition of the SUCC Function	171
251. Example of the SUCC Function	171
252. Definition of the TERMIN Procedure	171
253. Definition of the TERMOUT Procedure	172
254. Definition of the TOKEN Procedure	172
255. Example of the TOKEN Procedure	173
256. Definition of the TRACE Procedure	173
257. Definition of the TRIM Function	174
258. Example of the TRIM Function	174
259. Definition of the TRUNC Function	174
260. Example of the TRUNC Function	175
261. Definition of the UNPACK Procedure	175
262. Example of the UNPACK Procedure	176
263. Definition of the UPDATE Procedure	176
264. Example of the UPDATE Procedure	177
265. Definition of the USEHEAP Procedure	177
266. Definition of the WRITE Procedure (for Record Files)	177
267. Example of the WRITE Procedure for Record Files	178
268. Example of Multiple Expressions on WRITE	178
269. Definition of the WRITE and WRITELN Procedures (for Text Files)	179
270. Example of Multiple Expressions on WRITE and WRITELN	179
271. Default Field Widths on WRITE and WRITELN	180
272. Examples of Writing Boolean Data	181
273. Examples of Writing CHAR Data	181

274.	Examples of Writing DBCS Fixed String Data	181
275.	Examples of Writing GCHAR Data	182
276.	Examples of Writing GSTRING Data	182
277.	Examples of Writing Integer Data	182
278.	Examples of Writing Real Data	183
279.	Examples of Writing SBCS Fixed String Data	183
280.	Examples of Writing String Data	184
281.	Examples of Writing Mixed String Data	184
282.	Definition of the WRITESTR Procedure	185
283.	Example of the WRITESTR Procedure	185
284.	Example of Code Equivalents to WRITESTR	186
285.	Definition of the CMS Procedure	186
286.	Example of the CMS Procedure	187
287.	Definition of the ITOHS Function	187
288.	Example of the ITOHS Function	187
289.	Declaration of the ONERROR Procedure	188
290.	Definition of the PICTURE Function	189
291.	Examples of the PICTURE Function	191
292.	Syntax of VS Pascal Expressions	195
293.	Examples of Using Signs in Simple Expressions	196
294.	Examples of VS Pascal Expressions	196
295.	NOT Operators	197
296.	Multiplication Operators	197
297.	Addition Operators	198
298.	Relational Operators	199
299.	Example of a Boolean Expression	200
300.	Predefined Functions Permitted in Constant Expressions	201
301.	Examples of Constant Expressions	201
302.	Logical Operators for Integer Operands	202
303.	Examples of Logical Operations	202
304.	Syntax of a Function Call	203
305.	Example of a Function Call	203
306.	Syntax of an Ordinal Conversion	204
307.	Examples of the Ordinal Conversion Function	204
308.	Syntax of a Set Constructor	204
309.	Example of a Set Constructor	205
310.	Syntax of VS Pascal Statements	208
311.	Summary of VS Pascal Statements	208
312.	Syntax of the ASSERT Statement	209
313.	Example of the ASSERT Statement	209
314.	Syntax of the Assignment Statement	210
315.	Example of the Assignment Statement	211
316.	Syntax of the CASE Statement	212
317.	Example of the CASE Statement	213
318.	Example of the CASE Statement with the OTHERWISE Keyword	214
319.	Syntax of the Compound Statement	214
320.	Example of the Compound Statement	214
321.	Syntax of the CONTINUE Statement	215
322.	Example of the CONTINUE Statement and Its Equivalent	215
323.	Syntax of the Empty Statement	215
324.	Example of the Empty Statement	216
325.	Syntax of the FOR Statement	216
326.	Example of the Equivalent of a FOR-TO Statement	217
327.	Example of the Equivalent of a FOR-DOWNTO Statement	218
328.	Examples of the FOR Statement	218
329.	Syntax of the GOTO Statement	219

330.	Example of Using the GOTO Statement to Leave a Function	219
331.	Example of Valid and Invalid GOTO Statements	220
332.	Syntax of the IF Statement	220
333.	Examples of Simple IF Statements	221
334.	Example of Nested IF Statements	221
335.	Example of Nested IF Statements with the Empty Statement	221
336.	Syntax of the LEAVE Statement	222
337.	Example of the LEAVE Statement	222
338.	Example of the LEAVE Statement and Its Equivalent	222
339.	Syntax of the Procedure Call	223
340.	Example of Procedure Calls	223
341.	Syntax of the REPEAT Statement	224
342.	Example of the REPEAT Statement	224
343.	Syntax of the RETURN Statement	224
344.	Example of the RETURN Statement	224
345.	Syntax of the WHILE Statement	225
346.	Example of the WHILE Statement	225
347.	Syntax of the WITH Statement	225
348.	Example of the WITH Statement	226
349.	Example of WITH Statement Evaluation	226
350.	Example of Nested WITH Statement and Identifier Scoping	227
351.	Summary of VS Pascal Compiler Directives	230
352.	Syntax of the %CHECK Directive	232
353.	Syntax of the %CPAGE Directive	232
354.	Example of the %CPAGE Directive	233
355.	Syntax of the %ENDSELECT Directive	233
356.	Syntax of the %INCLUDE Directive	233
357.	Example of the %INCLUDE Directive	234
358.	Syntax of the %LIST Directive	234
359.	Syntax of the %MARGINS Directive	235
360.	Syntax of the %PAGE Directive	235
361.	Syntax of the %PRINT Directive	235
362.	Syntax of the %SELECT Directive	236
363.	Syntax of the %SKIP Directive	236
364.	Syntax of the %SPACE Directive	236
365.	Syntax of the %TITLE Directive	237
366.	Syntax of the %UHEADER Directive	237
367.	Example of the %UHEADER Directive	238
368.	Syntax of the %WHEN Directive	238
369.	Example of Conditional Compilation	239
370.	Syntax of the %WRITE Directive	240
371.	Predefined Identifiers	246
372.	Syntax of Options for Opening Files	252
373.	Exceptions in VS Pascal Release 2 Support of Release 1	270
374.	Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2	273

Chapter 1. How to Read Syntax Diagrams

Chapter 1. How to Read Syntax Diagrams

Read syntax diagrams from left to right, top to bottom.

- ▶— indicates the beginning of the diagram.
- ▶ indicates that the syntax is continued on the next line.
- ▶— indicates that the syntax is continued from the previous line.
- ▶ indicates the end of the diagram.

Keywords appear in all capital letters. For example: VAR, BEGIN, END. When writing code, enter keywords exactly as shown, either in all caps or in lowercase.

Variables appear in lowercase in a special typeface. For example: *label-dcl*.

Special Symbols such as ">", "=", and so forth must be entered as part of the code.

No Parameters

A keyword that requires no parameter is diagrammed this way:



Remember that you must code keywords (those in capital letters) exactly as shown, or in lowercase. In this example, you can code

STATEMENT

or

statement

Required Parameters

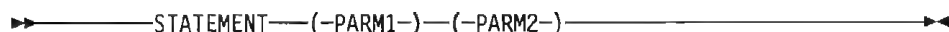
All required keywords and variables appear on the diagram's main path. In this example,



you must code both parameters. Always separate parameters with one or more blanks.

STATEMENT PARM1 PARM2

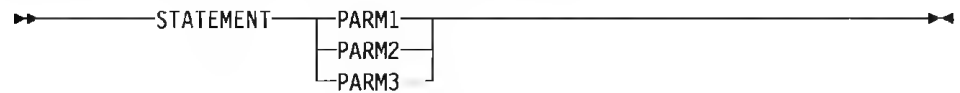
Parentheses around parameters, like all special symbols, must be coded exactly as shown. In this example,



you must code:

STATEMENT (PARM1) (PARM2)

When there is a vertical list of parameters, one of which is on the main path, you *must* choose only one of them. In this example,



you must code:

STATEMENT PARM1

or

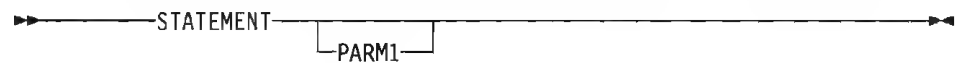
STATEMENT PARM2

or

STATEMENT PARM3

Optional Parameters

A single optional parameter appears below the main path. In this example,



you must code either:

STATEMENT

or

STATEMENT PARM1

When you can choose only one optional parameter from a list of two or more, the choices appear in a vertical list below the main path. In this example,



you must code:

STATEMENT

or

STATEMENT PARM1

or

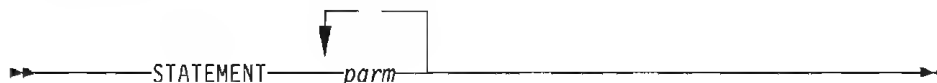
STATEMENT PARM2

Multiple Parameters

The repeat symbol



indicates that you can specify more than one parameter or a single parameter more than once. In this example,



parm, shown in small letters, represents a variable parameter. If the values you can substitute for *parm* include PARM1 and PARM2, then you can code:

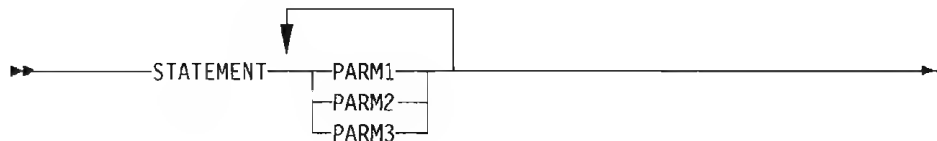
```
STATEMENT parm1
```

or

```
STATEMENT parm1 parm2
```

and so forth.

This diagram



indicates that you can code:

```
STATEMENT PARM1
```

or

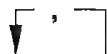
```
STATEMENT PARM1 PARM3
```

or

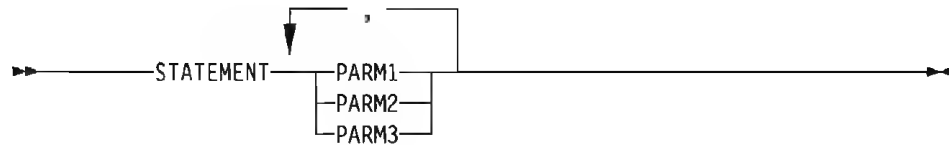
```
STATEMENT PARM1 PARM2 PARM3
```

and so forth.

When the repeat symbol contains a comma,



you must separate multiple parameters with commas. In such cases, parameters need not be separated by blanks. In this example,



you can code:

STATEMENT PARM1

or

STATEMENT PARM1, PARM3

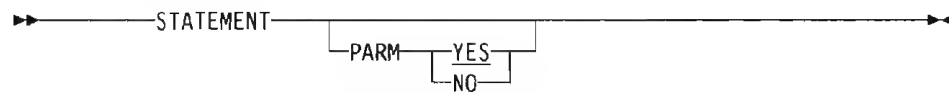
or

STATEMENT PARM1,PARM2,PARM3

and so forth.

Default Parameters

Default parameters are underscored. Omitting a parameter with a default value produces the same result as actually coding the default. In this example,



coding

STATEMENT PARM

is equivalent to coding

STATEMENT PARM YES

Chapter 2. VS Pascal Program Elements

Chapter 2. VS Pascal Program Elements

VS Pascal provides several features, or basic elements, that are found in most VS Pascal programs. The following sections define these program elements and discuss the conventions governing their use.

Identifiers

Identifiers are the names given to variables, data types, procedures, functions, named constants, and units. In VS Pascal, you can also associate an identifier with a label.

There are two types of identifiers in VS Pascal:

- Predefined identifiers, which VS Pascal supplies
- User-defined identifiers, which you supply.

VS Pascal identifiers must:

- Be no longer than 100 characters
- Start with a letter or dollar sign
- Be completely contained on one source code line.

Figure 1 shows the syntax of VS Pascal identifiers.

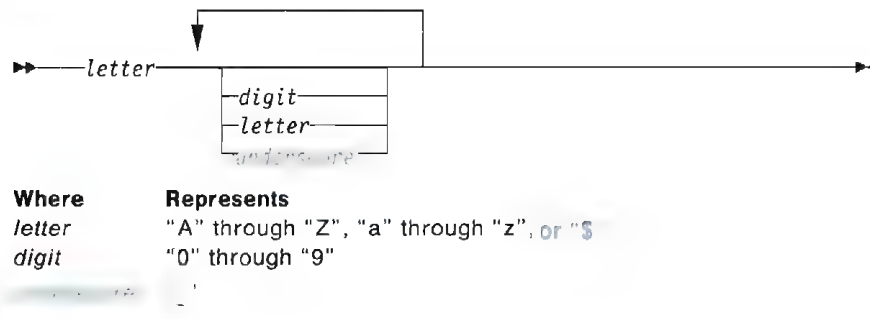


Figure 1. Syntax of a VS Pascal Identifier

Note to Figure 1: The dollar sign (\$) is assumed to be '\$B'xC

VS Pascal makes no distinction between lowercase and uppercase letters within an identifier. For example, the identifiers "ALPHA", "alpha", "Alpha", and even "alPha" are equivalent.

Figure 2 on page 9 shows examples of valid and invalid identifiers.

Valid Identifiers

I
K9
New_York
AMOUNT\$

Invalid Identifiers

5K (starts with a number)
NEW JERSEY (has a blank between the words)

Figure 2. Examples of Valid and Invalid Identifiers

Reserved Words

Reserved words are tokens that express the syntax of VS Pascal. You cannot redeclare reserved words for other uses. Reserved words must be separated from other reserved words and identifiers by a special symbol, a comment, or at least one blank. VS Pascal makes no distinction between uppercase and lowercase for reserved words.

Figure 3 lists the VS Pascal reserved words.

AND	END	OF	SPACE
ARRAY	FILE	OR	STATIC
ASSERT	FOR	OTHERWISE	THEN
BEGIN	FUNCTION	PACKED	TO
CASE	GOTO	PROCEDURE	TYPE
CONST	IF	PROGRAM	UNTIL
CONTINUE	IN	RANGE	VALUE
DEF	LABEL	RECORD	VAR
DIV	LEAVE	REF	WHILE
DO	MOD	REPEAT	WITH
DOWNT0	NIL	RETURN	XOR
ELSE	NOT	SET	

Figure 3. VS Pascal Reserved Words

Note to Figure 3: The reserved words highlighted in color are IBM extensions to Standard Pascal, and as such they are reserved only when the `LANGLVL(EXTENDED)` compile-time option is in effect. These words can be redefined when compiling Standard Pascal.

Special Symbols

Special symbols are nonalphabetic characters or groups of characters used to express VS Pascal syntax. Special symbols are used to represent such syntax elements as operators and variable qualifiers. Figure 4 describes the VS Pascal special symbols.

Symbol	Represents
+	Addition, concatenation, and set union operator
-	Subtraction and set difference operator
*	Multiplication and set intersection operator
/	Division operator, real result only
~	Boolean NOT operator, one's complement on integer, and set complement
	Boolean OR operator, and logical OR on integer
&	Boolean AND operator, and logical AND on integer
> ^ or &^	Boolean XOR operator, logical XOR on integer, and set symmetric difference
=	Equality operator
<	Less than operator
< =	Less than or equal to operator, and set subset operator
> =	Greater than or equal to operator, and set superset operator
>	Greater than operator
< > or ≠	Not equal operator
< < <	Left logical shift operator on integer
> > >	Right logical shift operator on integer
	Concatenation operator
:=	Assignment symbol
.	Period to end a unit, and a field separator in a record
,	Comma, used as a list separator
:	Colon, used to specify a definition
;	Semicolon, used as a statement separator
..	Subrange notation
'	Quote, used to begin and end string constants
@ or ->	Pointer symbol
(Left parenthesis, used for parameter lists and mathematical grouping
)	Right parenthesis, used for parameter lists and mathematical grouping
[or {.	Left square bracket, used for subscripts and set constructors

Figure 4 (Part 1 of 2). Summary of VS Pascal Special Symbols

Symbol	Represents
] or .)	Right square bracket, used for subscripts and set constructors
{ or (Comment left brace (standard)
} or)	Comment right brace (standard)
/*	Comment left brace (alternate form)
*/	Comment right brace (alternate form)

Figure 4 (Part 2 of 2). Summary of VS Pascal Special Symbols

The special symbols in Figure 5 can also be written as reserved words

Symbol	Reserved Word
¬	NOT
	OR
&	AND
> < or &&	XOR

Figure 5. Symbols That Are Reserved Words

Comments

Comments are written statements that annotate source code by explaining or describing some aspect of the code. Comments do not affect the execution of a program. A comment can be placed in a unit anywhere a blank is acceptable.

Because the compiler bypasses comments, you can use them to temporarily “comment out,” or exclude, code lines from compilation.

You must enclose comments with “{...}” or “/*...*/”. The compiler recognizes these as two different forms of comment. However, the compiler considers the symbols “(” and “)” to be identical to the left and right braces.

When the compiler encounters “{”, it bypasses all characters that follow until it encounters the “}” symbol. Likewise, the compiler bypasses all characters following “/*” until it encounters “*/”. As a result, either form can enclose the other. For example, the compiler considers /*...{...}...*/ to be one comment!

You can use the two different sets of comment delimiters to nest comments. You might, for example, use one comment form for ordinary comments and use the other to block out temporary sections of code (perhaps debugging statements), as illustrated in Figure 6 on page 12.

```

/*
IF A = 10 THEN          (* this statement is
                        for program
                        debugging only *)
    WRITE('A IS EQUAL TO TEN');
*/

```

Figure 6. Example of a Nested Comment

Note: Comments contained within a %WHEN block that evaluates to FALSE must not contain "%" followed by blanks and "when". The compiler will interpret this statement as the beginning of a new %WHEN block.

Note for MVS batch: Do not use a "/" comment delimiter in the first character position of a line.

Double-Byte Character Set (DBCS) Comments

Double-byte character set (DBCS) portions of comments must be enclosed by a beginning shift-out (X'0E') character and an ending shift-in (X'0F') character.

Restriction: DBCS portions of comments cannot span multiple lines.

VS Pascal recognizes the shift-out and shift-in characters only when the GRAPHIC compile-time option is in effect.

Figure 7 shows an example of mixed single-byte and double-byte portions of comments.

```

(*ABCDE<.A.B>FGHIJ*)

```

Figure 7. Example of DBCS Comments

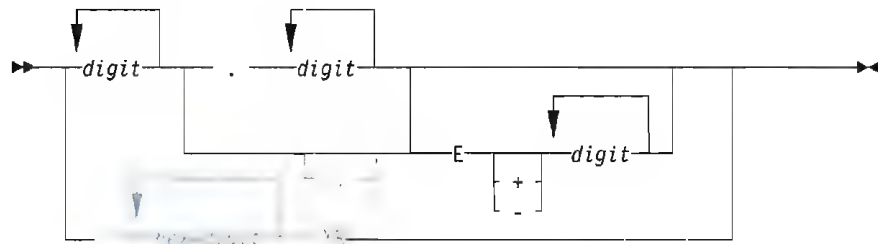
Literals

Literals are values that define and represent no other value but themselves. Literals are not assigned to represent other values. Figure 8 on page 13 defines the syntax of VS Pascal literals.

Unsigned Integer



Unsigned Real Number



String



DBCS String



Where	Represents
<i>binary-digit</i>	0 or 1
<i>digit</i>	"0" through "9"
<i>hex-digit</i>	0 through 9, A through F, and a through f
<i>character</i>	Any EBCDIC character
<i>DBCS-character</i>	Any DBCS character

Figure 8. Syntax of Literals

The symbol "E" or "e" when used in a real number expresses "ten to the power of."

If you want a single quote to be recognized in a string, then you must write the character twice. For example, for the string SQUARE'S_SIDES, you must type 'SQUARE' 'S_SIDES'.

VS Pascal is case sensitive with regard to the characters in string literals; uppercase and lowercase letters are considered different. Also, string literals written in VS Pascal cannot extend past the end of a line in the code. *If you must use a string literal that extends past the end of a line, then concatenate shorter strings together.* Figure 9 shows some examples of literals.

Constant Matches	Standard Type
0	INTEGER
-500	INTEGER
1.0	REAL
314159E-5	REAL
0E0	REAL
1.0E10	REAL
TRUE	BOOLEAN
'FF'X	INTEGER
'A'	CHAR
'<.A>'G	GCHAR
'ABC'	STRING
'<.A.B.C>'G	GSTRING
'C1C2C2'XC	STRING
'4E800000FFFFFFFF'XR	REAL
'42C142C2'XG	STRING
'abc'	STRING
' '	STRING
' ' ' '	CHAR
' ' ' '	CHAR
' ' ' ' ' '	STRING
'That' 's all'	STRING

Figure 9. Examples of Literals

Hexadecimal and Binary Literals

Integer Hexadecimal Literals: These are enclosed in single quotes and suffixed with an "X" or "x"—for example, 'FF'X. Hexadecimal literals can be used anywhere an integer literal is appropriate. If you specify fewer than eight digits (4 bytes), VS Pascal assumes the digits not supplied are zeros on the left. For example, 'F'X is the value 15.

Integer Binary Literals: These are enclosed in single quotes and suffixed with a "B" or "b"—for example, '00000110'b. Binary literals can be used anywhere an integer literal is appropriate. If you specify fewer than 32 binary digits (4 bytes), VS Pascal assumes the digits not supplied are zeros on the left. For example, '1111'B is the value 15.

Floating-Point Hexadecimal Literals: These are enclosed in quotes and suffixed with an "XR" or "xr". Such literals can be used anywhere a real literal is appropriate. If you specify fewer than 16 hexadecimal digits (8 bytes), VS Pascal assumes the digits not supplied are zeros on the right. For example, '4110'xr is the same as '4110000000000000'xr.

String Hexadecimal Literals: These are enclosed in quotes and suffixed with an "XC" or "xc". Such literals can be used anywhere a string literal is appropriate. All characters in the string must be specified fully; that is, there must be an even number of digits. For example, 'C1C2C2'XC is a valid string hexadecimal literal, but 'C1C12' generates an error, because the compiler cannot make assumptions about the placement of the missing character. DBCS validity checking is not done on string hexadecimal literals.

DBCS Hexadecimal Literals: These are enclosed in quotes and suffixed with an "XG" or "xg". VS Pascal performs a code point check to ensure that the code ranges of all DBCS hexadecimal literals are from '41'X to 'FE'X for either the first or second byte, or '4040'X. VS Pascal also performs a length check to ensure that DBCS hexadecimal literal strings are a multiple of four. VS Pascal considers anything other than a multiple of four to be an error, because it cannot make assumptions about the placement of any missing digits. An example of a valid DBCS hexadecimal literal is 'H1H2H3H4'XG, where 'H1' is a single hexadecimal digit. Shift-out and shift-in characters are not needed in DBCS hexadecimal string literals.

Double-Byte Character Set (DBCS) Literals

These are enclosed in quotes and suffixed with a "G" or "g". Within the quotes, there must be only a shift-out character, a string of DBCS characters, and a shift-in character. Figure 10 shows an example of a DBCS literal.

```
'<.A.B.C.D.E.F>'G
```

Figure 10. Example of a DBCS Literal

Double-Byte Character Set (DBCS) Mixed Literals

VS Pascal character strings can contain both DBCS characters and single-byte character set (SBCS) characters in what is called a *DBCS mixed literal*.

DBCS portions of the string must be enclosed by a beginning shift-out (X'0E') character and an ending shift-in (X'0F') character. VS Pascal operators and most predefined routines manipulate DBCS strings in a byte-oriented manner.

Figure 11 on page 16 shows an example of a DBCS mixed literal.

'ABCDE < .A.B > FGHIJ'

Figure 11. Example of a DBCS Mixed Literal

Figure 12 shows the syntax of valid mixed strings.

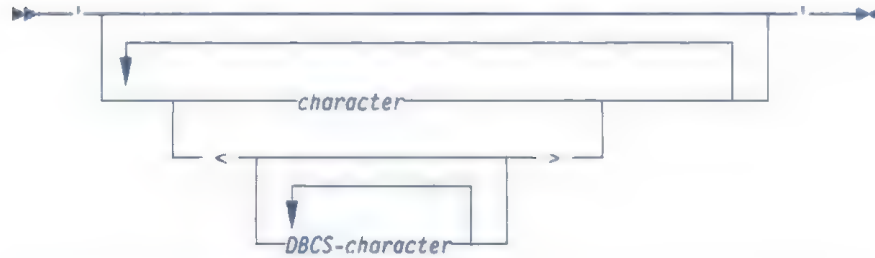


Figure 12. Syntax of Valid Mixed Strings

Figure 13 shows the syntax of canonical mixed strings.

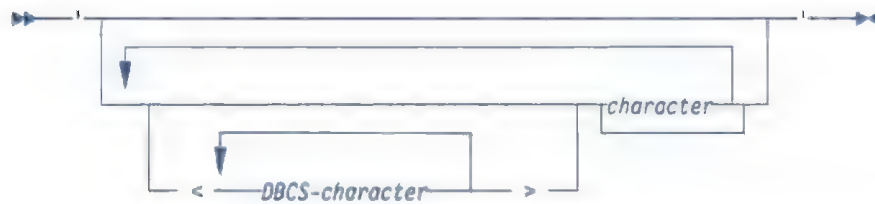


Figure 13. Syntax of Canonical Mixed Strings

Chapter 3. Structure of VS Pascal Programs

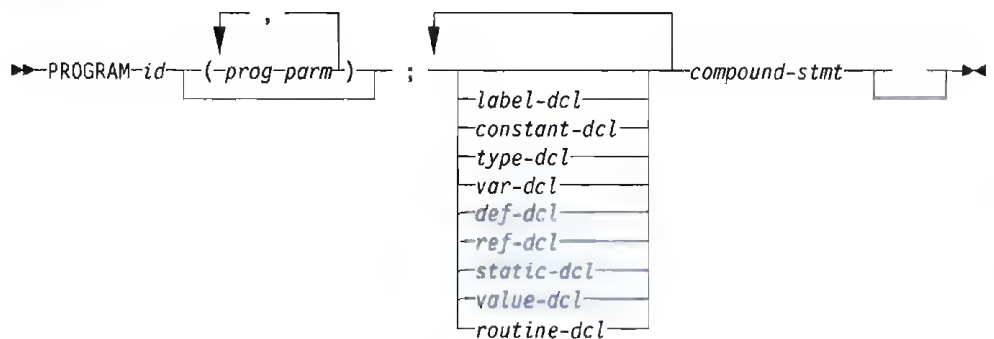
Chapter 3. Structure of VS Pascal Programs

All VS Pascal programs are comprised of separately compilable units that are link-edited together at run time. A *compilable unit*, or simply *unit*, consists of declarations (such as data, procedure, and function declarations) and statements. Because a unit can be compiled independently from the rest of the program, programmers can logically organize their code. There are two types of units in VS Pascal: the program unit and the segment unit.

Program Units

The *program unit* initially gains control when you invoke a compiled program from the operating system. It is a procedure invoked by the operating system. A program unit must contain the keyword PROGRAM. Figure 14 shows the syntax of a program unit.

Program Unit



Where	Represents
PROGRAM	Keyword
id	A unique external name
prog parm	An optional list of program parameters that specify links to external names
dcl	A LABEL, CONST, TYPE, VAR, DEF, REF, STATIC, VALUE, or routine declaration
compound-stmt	A compound statement

Figure 14. Syntax of a Program Unit

Figure 15 on page 19 shows the structure of a program unit. The various kinds of declarations in VS Pascal are optional **and can appear in any order**. The only required items are the program header and the main program block.

PROGRAM HEADER;	Program Header
LABEL DECLARATIONS;	Label Declarations
CONSTANT DEFINITIONS; TYPE DEFINITIONS; VARIABLE DECLARATIONS;	Data Descriptions
PROCEDURE DECLARATIONS; FUNCTION DECLARATIONS;	Routine Declarations
BEGIN STATEMENTS; . . . STATEMENTS; END.	Main Program Block

Figure 15. Structure of a Program Unit

Figure 16 shows an example of a program unit.

```

PROGRAM EXAMPLE;
VAR
  I : INTEGER;
BEGIN
  FOR I:=0 TO 1000 DO
    IF I MOD 7 = 0 THEN
      WRITELN( I:5,
        ' IS DIVISIBLE BY SEVEN');
  END.

```

Figure 16. Example of a Program Unit

The program parameter list must specify all external bindings with Pascal variables. There are separate considerations for Standard and VS Pascal.

Program Parameters For Standard Pascal:

- If your program uses the predefined files **INPUT** and **OUTPUT**, you must specify them in the program parameter list.
- Specifying **INPUT** as a program parameter opens that file for **INPUT(RESET)**.
- Specifying **OUTPUT** as a program parameter opens that file for **OUTPUT(REWRITE)**.
- If you specify **INPUT** or **OUTPUT** as program parameters, you cannot redefine them as global variables in your program.

- You cannot specify duplicate identifiers in the program parameter list. For example,

```
PROGRAM USER(OUTPUT,F,F);
```

is invalid because F is specified twice.

- Any identifier specified in the program parameter list (except INPUT and OUTPUT) must be defined as a variable identifier in the program block. For example,

```
PROGRAM USER(OUTPUT,F,G);
CONST
  G = 3;
VAR
BEGIN
.
.
END.
```

is invalid because G is declared as a constant rather than as a variable and F is not declared at all.

Program Parameters For VS Pascal: All of the program parameters for Standard Pascal also apply to VS Pascal, with these exceptions:

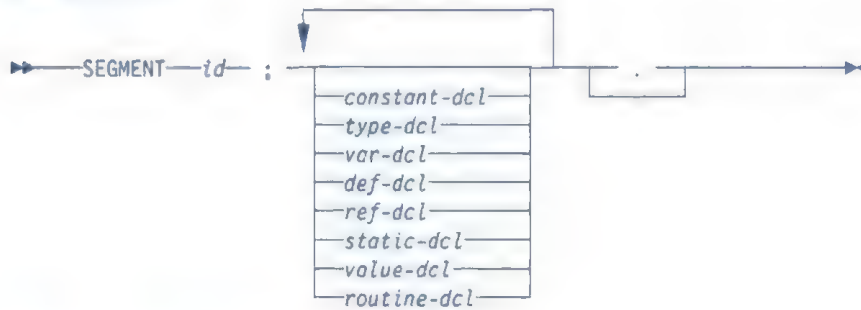
- INPUT and OUTPUT are predefined, and therefore you need not specify them in the program parameter list when they appear in a program.
- If you want, you can redefine INPUT and OUTPUT in your program.

Note: Because program parameters are used to specify external bindings, their identifiers must conform to the same rules as external identifiers of the type specified on the program parameter's variable declaration.

Segment Units

A *segment unit* is any unit that can be compiled independently of the program unit. It consists of routines to be linked into the final program before execution. Segments are useful in breaking up large VS Pascal programs into smaller units. Figure 17 on page 21 shows the syntax of a segment unit.

Segment Unit



Where	Represents
SEGMENT	Keyword
id	A unique external name
dcl	A CONST, TYPE, VAR, DEF, REF, STATIC, VALUE, or routine declaration

Figure 17. Syntax of a Segment Unit

The various kinds of declarations in VS Pascal are optional and can appear in any order. The only required item is the segment header. Figure 18 shows the structure of a segment unit.

SEGMENT HEADER;	Segment Header
CONSTANT DEFINITIONS;	Data Descriptions
TYPE DEFINITIONS;	
VARIABLE DECLARATIONS;	
PROCEDURE DECLARATIONS;	Routine Declarations
FUNCTION DECLARATIONS;	

Figure 18. Structure of a Segment Unit

Data is passed to routines through parameters and external variables. A segment unit can access the global automatic variables of the program unit (see "VAR Declaration" on page 31 for more information). Figure 19 shows an example of a segment unit.

```
SEGMENT COSINE;  
FUNCTION COSINE (X : REAL ) : REAL; EXTERNAL;  
FUNCTION COSINE;  
VAR  
    S : REAL;  
BEGIN  
    S := SIN(X);  
    COSINE := Sqrt(1.0 - S*S);  
END; .
```

Figure 19. Example of a Segment Unit

Linking Units to Form a Program

A VS Pascal program is formed by linking a program unit to:

- Segment units (if any)
- The VS Pascal run-time library
- Any libraries that you might supply.

Figure 20 illustrates the relationship between program and segment units, the VS Pascal run-time library, and additional user-supplied libraries.

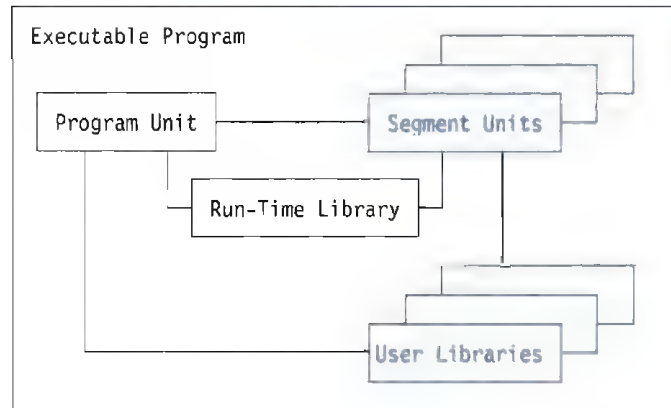


Figure 20. Linking a Program Unit with Segment Units

Chapter 4. Declarations

Declarations associate names with objects, such as data types, variables, and routines, that are used in a program.

Figure 21 summarizes the types of declarations available in VS Pascal. The rest of this chapter provides descriptions of each declaration type in alphabetic order. The PROCEDURE and FUNCTION declarations are discussed in Chapter 8, "Routines" on page 102.

Declaration Type	Function	See Page
LABEL	Declares labels to be referenced by a GOTO statement	28
CONST	Assigns identifiers to be used as synonyms for constant expressions	27
TYPE	Declares data types	30
VAR	Declares automatic variables (those allocated when a routine is invoked and then deallocated when the routine is exited)	31
DEF	Declares external variables	28
REF	Declares external variables	29
STATIC	Declares static variables	30
VALUE	Specifies initial values for static and DEF variables	31
PROCEDURE	Defines routines that may or may not pass a result back to the invoker (discussed in Chapter 8, "Routines" on page 102)	102
FUNCTION	Defines a routine that passes a result back to the invoker through the routine name (discussed in Chapter 8, "Routines" on page 102)	102

Figure 21 Summary of VS Pascal Declarations

The syntax of VS Pascal requires that all identifiers be predefined or declared before you can use them. There is one exception to this rule: a pointer definition can refer to an identifier before that identifier is declared. The identifier must be declared later, or VS Pascal generates a compiler diagnostic message.

Lexical Scope of Identifiers

The *lexical scope*, or simply *scope*, of an identifier is the area of a unit where the identifier can be referenced. The scope of an identifier can be either local or global:

- A *local identifier* is an identifier declared in a function or procedure. A local identifier has no effect on an outside function or procedure.
- A *global identifier* is an identifier declared in the main program. A global identifier can be used, referenced, or changed anywhere in the program, including any functions or procedures.

For example, in Figure 22 on page 25, any identifier declared in Program A will be global. If Program A declares Procedure B and Function C, Procedure B and

Function C can reference the global identifiers declared in Program A. If Procedure B declares an identifier, it will be a local identifier to Procedure B, and Program A or Function C cannot access that identifier.

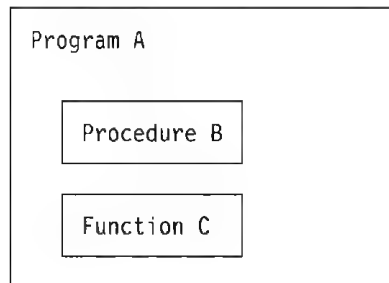
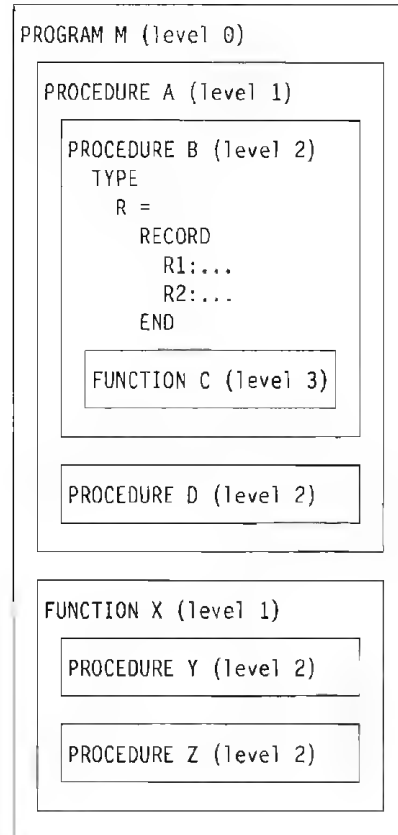


Figure 22. Scope of identifiers

The scope of any particular identifier depends on the structure of the routine declarations within the unit in which it appears. An identifier's scope extends to the entire routine (or unit) in which it is declared, including all other routines nested within the routine. Because routines can be nested within other routines, each routine has an associated *lexical level*. Record declarations also define a lexical scope for their fields.

Within a lexical level, an identifier can be defined only once. A program unit is at lexical level 0, routines defined within level 0 are at level 1, and so forth. In general, a routine defined in level i is at level $(i + 1)$.

Figure 23 on page 26 illustrates a nesting structure.



Identifiers Declared In	Are Accessible In
PROGRAM M	M, A, B, R, C, D, X, Y, Z
PROCEDURE A	A, B, R, C, D
PROCEDURE B	B, R, C
TYPE R	B, C (see note)
FUNCTION C	C
PROCEDURE D	D
FUNCTION X	X, Y, Z
PROCEDURE Y	Y
PROCEDURE Z	Z

Figure 23. Nesting Structure of a Program

Note to Figure 23: The scope of a field identifier is limited to the scope of the record in which it is defined. The field identifiers of a record can be accessed either by referencing the field or by using the WITH statement.

When an identifier is declared within the scope of an existing identifier of the same name, the new identifier is the one recognized when the name appears in the routine. For example, in Figure 23, function C is nested in procedure B, procedure B is nested in procedure A, and procedure A is nested in program M. If both program M and procedure B declared an identifier X, a conflict could arise. The conflict is resolved by using the most recent declaration of X. Thus, in program M and procedure A, the identifier X declared in program M would be used, while in procedure B and function C, the identifier X declared in procedure B would be used.

The VS Pascal compiler inserts a prelude of declarations at the beginning of every unit it compiles. These declarations consist of predefined types, constants, routines, and variables. See Appendix B, “Predefined Identifiers” on page 246 for a list of all predefined identifiers. The scope of the prelude encompasses the entire unit. You can, if you want, redeclare any of these predefined identifiers, although this is not recommended.

Declaration Order

Standard Pascal declarations must be made in this order:

1. LABEL
2. CONST
3. TYPE
4. VAR
5. PROCEDURE
6. FUNCTION

VS Pascal declarations can be made in any order, but this is the typical order:

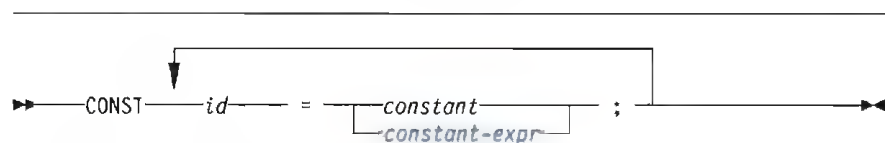
1. LABEL
2. CONST
3. TYPE
4. VAR
5. DEF
6. REF
7. STATIC
8. VALUE
9. PROCEDURE
10. FUNCTION

Because VS Pascal allows declarations to be made in any order, source code included during compilation can be independent of any ordering already established in the unit.

VS Pascal Declarations

CONST Declaration

CONST assigns identifiers to be used as synonyms for constant expressions. The type of a constant identifier is determined by the type of the expression in the declaration. Figure 24 shows the syntax of the CONST declaration.



Where	Represents
CONST	The Standard Pascal keyword
<i>id</i>	An identifier assigned to a constant or constant expression
<i>constant</i>	Any constant
<i>constant-expr</i>	Any constant expression

Figure 24. Syntax of the CONST Declaration

See Figure 32 on page 33 for an example of a CONST declaration.

DEF Declaration

DEF declares *external variables* that are allocated before execution and that can be accessed from more than one unit. When several units declare an external variable with the same name, the program loader associates a single common storage location with that variable name. Figure 25 shows the syntax of a DEF declaration.

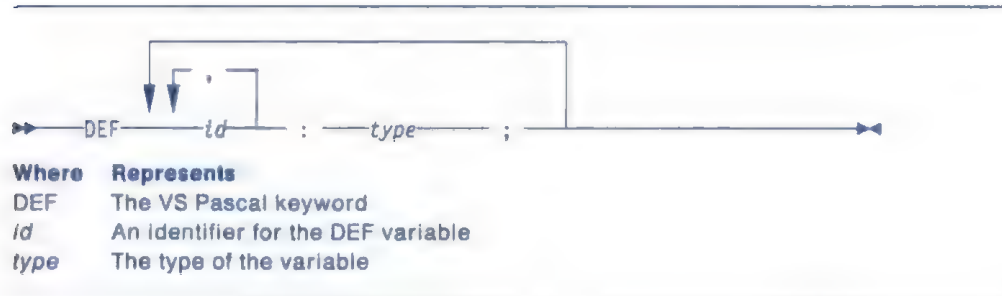


Figure 25. Syntax of the DEF Declaration

DEF variable names are governed by the same scoping rules that apply to other declared identifiers. However, when a DEF variable name in one scope or unit matches an external variable name in another scope or unit, both occurrences of the variable reference the same storage.

Data in external variables that are local to a routine will be preserved over separate calls to the routine. Recursive calls to such a routine access the same value of the external variable.

DEF variables can be initialized at compile time with a VALUE declaration. See "VALUE Declaration" on page 31.

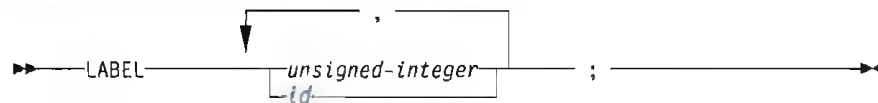
Restrictions:

- Programs that modify DEF variables are not reentrant.
- External variables of the same name must have identical data types in all units. You must ensure that the types are the same; the compiler cannot verify this.
- External variables whose names have the first eight characters in common are allocated the same storage location. If you want two external variables to occupy different storage locations, you must ensure that the first eight characters of their names are unique; the compiler cannot verify this.

See Figure 32 on page 33 for an example of a DEF declaration.

LABEL Declaration

LABEL declares labels that are referenced by a GOTO statement within a routine. All labels defined within a routine must be declared in a LABEL declaration within the routine. Figure 26 on page 29 shows the syntax of the LABEL declaration.



Where	Represents
LABEL	The Standard Pascal keyword
<i>unsigned-integer</i>	A name assigned to a label which must be in the range 0 to 9999
<i>id</i>	A name assigned to a label

Figure 26 Syntax of the LABEL Declaration

See Figure 32 on page 33 for an example of a LABEL declaration.

REF Declaration

REF declares *external variables* that are allocated before execution and that can be accessed from more than one unit. Storage for the variables is defined in another unit. If the same external variable name is declared in several units, the program loader associates a single common storage location with that variable name. Figure 27 shows the syntax of the REF declaration.



Where	Represents
REF	The VS Pascal keyword
<i>id</i>	An identifier for the REF variable
<i>type</i>	The type of the variable

Figure 27. Syntax of the REF Declaration

REF variables are governed by the same scoping rules that apply to other declared identifiers. However, when a REF variable name in one scope or unit matches an external variable name in another scope or unit, both occurrences of the variable reference the same storage.

Data in external variables that are local to a routine will be preserved over separate calls to the routine. Recursive calls to such a routine access the same value of the external variable.

REF-declared variables remain unresolved until the encompassing unit is linked with a unit in which the variable is declared. The variable must be declared as a DEF variable, defined in a non-Pascal CSECT, or declared in an assembler language COM. Use REF variables to access external data declared in non-VS Pascal programs, such as those written in assembler language.

Restrictions:

- Programs that modify REF variables are not reentrant.
- External variables of the same name must have identical data types in all units. You must ensure that the types are the same; the compiler cannot verify this.

- External variables whose names have the first eight characters in common are allocated the same storage location. If you want two external variables to occupy different storage locations, you must ensure that the first eight characters of their names are unique; the compiler cannot verify this.

See Figure 32 on page 33 for an example of a REF declaration.

STATIC Declaration

STATIC declares *static variables* that are variables whose memory is allocated before execution of the program. This memory allocation exists while the program is being executed. Figure 28 shows the syntax of the STATIC declaration.



Where	Represents
STATIC	The VS Pascal keyword
id	An identifier for a static variable
type	The type of the variable

Figure 28. Syntax of the STATIC Declaration

Static variable names are governed by the same scoping rules that apply to other declared identifiers. Even when two static variables in different scopes have the same name, VS Pascal considers them to be different variables.

Data in static variables that are local to a routine will be preserved over separate calls to the routine. Recursive calls to such a routine access the same value of the static variable.

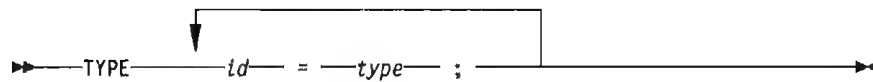
Static variables can be initialized at compile time with the VALUE declaration. See "VALUE Declaration" on page 31.

Restriction: Programs that modify static variables are not reentrant.

See Figure 32 on page 33 for an example of a STATIC declaration.

TYPE Declaration

TYPE defines a data type and associates a name with that type. Once declared, such a name can be used in the same way as a predefined data type name. Figure 29 on page 31 shows the syntax of the TYPE declaration.



Where	Represents
TYPE	The Standard Pascal keyword
<i>id</i>	An identifier for a type
<i>type</i>	The type

Figure 29. Syntax of the TYPE Declaration

See Figure 32 on page 33 for an example of a TYPE declaration.

VALUE Declaration

VALUE specifies initial values for static and DEF variables. The declarations consist of assignment statements, separated by semicolons, in the same form as the assignment statements in the body of a routine, with one exception—all subscripts and expressions must be constant expressions. See Chapter 5, "Constants" on page 36 for more information on constants.

Figure 30 shows the syntax of the VALUE declaration.



Where	Represents
VALUE	The VS Pascal keyword
<i>variable</i>	A variable to be assigned a value
<i>constant-expr</i>	Any constant expression
<i>structured-const</i>	Any structured constant

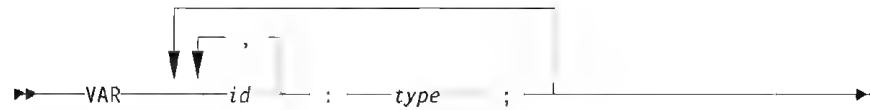
Figure 30. Syntax of the VALUE Declaration

Once a DEF variable is initialized with a VALUE declaration, it cannot be reinitialized with VALUE, either in the same unit or in a different unit. Although the compiler does not check for this violation, the system loader generates a diagnostic message when you combine units into a single load module. A static variable can be initialized with a VALUE declaration only once in a routine.

See Figure 32 on page 33 for an example of a VALUE declaration.

VAR Declaration

VAR declares *automatic variables* that are allocated when the routine in which they occur is invoked. The variables are deallocated when the corresponding return is made. Figure 31 on page 32 shows the syntax of the VAR declaration.



Where	Represents
VAR	The Standard Pascal keyword
id	An identifier for a VAR variable
type	The type of the variable

Figure 31. Syntax of the VAR Declaration

When a routine containing VAR declarations is invoked a second time before the initial invocation is complete (for example, in a recursive call), the local automatic variables are allocated again in a stack-like manner. The variables allocated for the first invocation become inaccessible until the recursive call is completed.

Variables should be declared as DEF variables if they are to be accessed across units. (See "DEF Declaration" on page 28.) However, if reentrancy is required, VS Pascal provides a method of sharing variables across units that does not rely on storage allocated at link-edit time.

Global automatic variables are those variables declared with VAR in the outermost nesting level of the main program. The global automatic variables of the main program can be accessed from a segment unit. Automatic variables declared in the outermost level of a segment are mapped directly on top of the main program's global variables. Therefore, to access the main program's global variables, a segment unit must have an identical copy of the main program's variable declarations. This mechanism is not as safe or as convenient as using DEF variables.

Note: Unpredictable errors can occur when the variables declared in a segment do not match those in the associated main program. The compiler has no way of checking their integrity. Use the %INCLUDE compiler directive to insert identical copies of the variables' declarations in all units (see "%INCLUDE Directive" on page 233).

Caution: Storage overlays can occur if you link units that contain global automatic variables to units written in different languages.

See Figure 32 on page 33 for an example of a VAR declaration.

Figure 32 on page 33 combines examples of the LABEL, CONST, TYPE, VAR, DEF, REF, STATIC, and VALUE declarations.

```

PROGRAM DATAEXAMPLES(INPUT, OUTPUT, TRACEFILE);

LABEL
  10,
  LABELA;

CONST
  ONEBLANK = ' ';
  LOTSOFBANKS = ' ';
  FIFTY = 50;
  A = FIFTY;
  B = FIFTY * 10/(3+2);
  CSQUARED = A*A + B*B;
  ORDOFZ = ORD('Z');
  MASK = '8000'X | '0400'X;
  ALPHALEN = 16;
  LETTERS = ['A'..'Z', 'a'..'z'];

TYPE
  CARDVALUE = 1..13;
  CARDSUIT = (SPADE, HEART, CLUB, DIAMOND);
  CARDTYPE = RECORD
    RANK : CARDVALUE;
    SUIT : CARDSUIT;
    FACEUP : BOOLEAN;
  END;
  GAMEHAND = ARRAY[1..13] OF CARDTYPE;

VAR
  I : INTEGER;
  TRACEFILE : TEXT;
  X,Y,Z : REAL;
  FLIP : RECORD
    DENOMINATION : (PENNY, NICKEL, DIME, QUARTER);
    UPSIDE : (HEADS, TAILS);
  END;

DEF
  C : INTEGER;
  R : REAL;
  MYHAND : GAMEHAND;

BEGIN
  10 :
    .
    .
    LABELA :
    .
    .
END.

```

Figure 32 (Part 1 of 2). Examples of VS Pascal Declarations

```

SEGMENT S;

CONST
    PI = 3.14159265358;

VAR
    I : INTEGER;
    TRACEFILE : TEXT;
    X,Y,Z : REAL;
    FLIP : RECORD
        DENOMINATION : (PENNY, NICKEL, DIME, QUARTER);
        UPSIDE : (HEADS, TAILS);
    END;

REF
    C : INTEGER;
    R : REAL;
    MYHAND : GAMEHAND;

STATIC
    J : INTEGER;
    K,
    L : REAL;

VALUE
    J := 17;
    K := PI;
    L := PI / 6;

PROCEDURE P;
BEGIN
    .
    .
END; .

```

Figure 32 (Part 2 of 2). Examples of VS Pascal Declarations

Chapter 5. Constants

Chapter 5. Constants

VS Pascal uses a number of different constants. The following sections explain what constants are and discuss the various types of constants.

Types of Constants

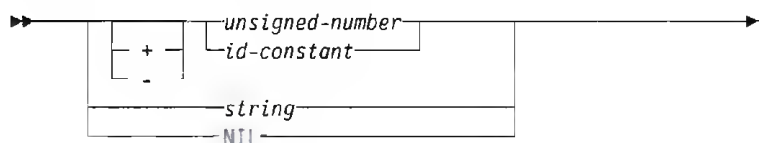
Constants are values that are either literals or identifiers associated with literals (*unsigned constants*) in CONST declarations. Figure 33 shows the categories into which constants can be grouped according to their predefined type.

Category	Predefined Type
Unsigned integers	Conform to types REAL, SHORTREAL , or INTEGER.
Strings	Conform to types STRING , SBCS fixed string, GSTRING , or DBCS fixed string. Note: A string one character long conforms to type CHAR or GCHAR .
TRUE and FALSE	Predefined in the language and are of type BOOLEAN.
NIL	A special type that conforms to any pointer type. Note: NIL represents a unique pointer value that is not a valid address.

Figure 33. Categories of Constants

Figure 34 defines the syntax of VS Pascal constants and unsigned constants.

Constant



Unsigned Constant



Unsigned Number

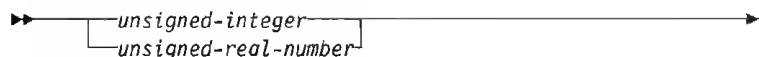


Figure 34. Syntax of Constants

Note to Figure 34: In "Constant," if the constant identifier has a sign, the identifier must represent a numeric value.

VS Pascal permits *constant expressions* in places where Standard Pascal permits only constants. Constant expressions are evaluated and replaced by a single result at compile time. See "Constant Expressions" on page 201 for more information.

Predefined Constants

Predefined constants are identifiers that are already defined within VS Pascal. The compiler inserts a prelude of declarations at the start of every compilable unit; the predefined constants are among the declarations, so there is no need to define these identifiers. (Although it is not recommended, you can redefine these identifiers if you prefer.)

ALFALEN	Length of type ALFA; value is 8.
ALPHALEN	Length of type ALPHA; value is 16.
EPSREAL	Smallest number such that $1.0 + \text{EPSREAL} > 1.0$: '3310000000000000'XR.
FALSE	Constant of type BOOLEAN; $\text{FALSE} < \text{TRUE}$.
MAXINT	Maximum value of type INTEGER: 2147483647.
MAXCHAR	Maximum value of type CHAR: 'FF'XC.
MAXREAL	Maximum value of type REAL: '7FFFFFFFFFFFFFFF'XR.
MININT	Minimum value of type INTEGER: -2147483648.
MINREAL	Minimum nonzero value of type REAL: '0010000000000000'XR.
TRUE	Constant of type BOOLEAN; $\text{TRUE} > \text{FALSE}$.

Structured Constants

Structured constants provide a convenient means of specifying a structured data element. The type of the constant is determined by the type identifier in the constant's definition. Structured constants can be used in value declarations, other constant declarations, or in executable expressions.

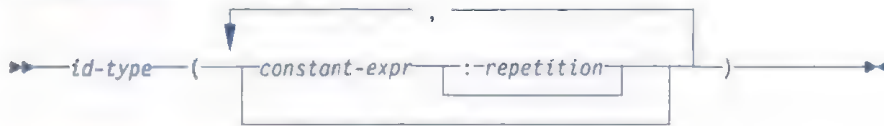
Restriction: A type containing a file cannot be used as the type name of a structured constant.

The syntax of structured constants is illustrated in Figure 35 on page 38.

Structured constant



Array structure



Record structure



Repetition (must evaluate to a positive integer)



Where	Represents
<i>constant-expr</i>	Any constant expression
<i>id-type</i>	An array or record type that does not contain a file

Figure 35. Syntax of Structured Constants

There are two types of structured constants: array constants and record constants.

Array Constants: These are specified by a list of constant expressions, each expression defining one element of the array. For a description of constant expressions, see "Constant Expressions" on page 201.

To omit an element in the middle of an array constant, you specify nothing between the commas. You can omit an element either within the list or at the end of the array. In either case, the value of that element is not defined. You can follow the constant expression with a colon and a repetition expression, indicating that the constant expression's value will be placed in the specified number of array elements. Figure 36 on page 39 shows an example of an array constant.

```

TYPE
  VECTOR = ARRAY[1..7] OF INTEGER;
  TETRA  = ARRAY[1..3,1..2,1..4] OF INTEGER;

CONST
  (*Structured Constants *)
  VECTOR_1  = VECTOR(7,0:5,1);
  VECTOR_2  = VECTOR(2,3,,4);
  ZERO_TETRA = TETRA(((0:4):2),
                     ((0:4), (0:4)),
                     ((0,0,0,0), (0,0,0,0)));

```

Figure 36. Example of an Array Constant

Record Constants: These are specified by a list of constant expressions, each expression defining one field of the record in the order declared. You can omit a field of the record within the list by specifying nothing between two commas, in which case the value of that field is not defined. Figure 37 shows an example of a record constant.

```

TYPE
  COMPLEX = RECORD
    RE, IM: REAL
  END;

CONST
  (*Structured Constants *)
  THREEFOUR = COMPLEX(3.0,4.0);

```

Figure 37. Example of a Record Constant

Values within the list can correspond to fields of a record's variant part. The tag field value must be specified immediately before the values to be assigned to the variant fields. (See Figure 38 on page 40.) When only a tag type is specified, the tag field must be specified even though it does not exist as a field. If the tag field is a back reference tag field (see Figure 73 on page 73), it must be specified twice in the list: once to be assigned a value, and again to identify the variant being referenced. If these two tag field specifications do not match, unpredictable results can occur.

Figure 38 on page 40 shows examples of structured constants with variant record fields.

```

TYPE
  FORM = (FCHAR, FINTEGER, FREAL, FSTRING);
  KONST =
    RECORD
      SIZE : INTEGER;
      CASE F : FORM OF
        FCHAR : (C : CHAR);
        FINTEGER : (I : INTEGER);
        FREAL : (
          CASE SIZE : OF
            4 : (S : SHORTREAL);
            8 : (R : REAL));
        FSTRING : (
          CASE BOOLEAN OF
            TRUE: (
              LEN : PACKED 0..32767;
              A : ALPHA);
            FALSE: (ST : STRING(16)));
    END;

CONST
  A = KONST(1,FCHAR,'A');
  INT = KONST(4,FINTEGER,3);
  SHORT = KONST(4,FREAL,4,1.2345);
  PI = KONST(8,FREAL,8,3.14159);
  STARS = KONST(4,FSTRING,TRUE,4,'*****');
  BARS = KONST(4,FSTRING,FALSE,'----');

```

Figure 38. Examples of Structured Constants with Variant Record Fields

The type identifier that begins a structured constant can be omitted if the structured constant is imbedded within another structured constant. This simplifies the syntax for structured constants that are multidimensional arrays or records with structured fields.

Figure 39 on page 41 shows an example of an array and a record constant combined.

```
TYPE
  COMPLEX = RECORD
    RE,IM: REAL
  END;
  CARRAY = ARRAY[0..9] OF COMPLEX;

CONST

  (*The following two declarations
   are equivalent *)
  VECTOR_3 = CARRAY(
    COMPLEX(1.0,0.0),
    COMPLEX(1.0,1.0):8,
    COMPLEX(0.0,0.0));
  VECTOR_4 = CARRAY(
    (1.0,0.0),
    (1.0,1.0):8,
    (0.0,0.0));
```

Figure 39. Example of a Combination of an Array and Record Constant

Chapter 6. Data Types

VS Pascal requires that every variable be declared and assigned a *data type*, or *type*, before it is used. The compiler checks each variable to ensure correct usage. This enables the programmer and the compiler to easily detect and prevent errors.

The Basic Data Types

Variables can be assigned one of four basic data types:

- Simple
- Pointer
- **String pointer**
- Structured.

Simple Data Types

Boolean	A true or false value
Character	All the values of the EBCDIC character set
DBCS Character	All the values of the double-byte character set (DBCS)
Enumerated	An ordered set of values defined by listing the identifiers that denote the values; for example, in an enumerated type, you can specify the days of the week to be Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday
Integer	A positive or negative whole number
Real	A positive or negative double-precision floating-point number
Shortreal	A positive or negative single-precision floating-point number
Subrange	A subset of a previously defined type formed by specifying the minimum and maximum allowable values.

Any of these simple data types can fall under one, or both, of the following general categories:

Scalar	A type whose values contain only one element; all simple data types are scalars
Ordinal	A scalar type whose values are mapped to a continuous range of integers; all scalars are ordinal except real, shortreal , and DBCS characters

Pointer Data Type

A pointer data type is used to reference a dynamic variable. A dynamic variable is a variable whose storage is allocated at run time.

String Pointer Data Type

A string pointer data type defines a pointer to a dynamic string variable. The maximum length of a string pointer is determined at run time.

Structured Data Types

Array	An indexed list of elements of the same data type
DBCS String	An array of DBCS characters whose length varies at run time up to a compile-time specified maximum
File	A sequence of components of the same data type
Record	A named list of fields that can be of different data types
Set	A collection of objects of an ordinal type
Space	A variable whose components can be positioned at any byte in the total space of the variable
String	An array of characters whose length varies at run time up to a compile-time specified maximum

Figure 40 shows the syntax of a data type.

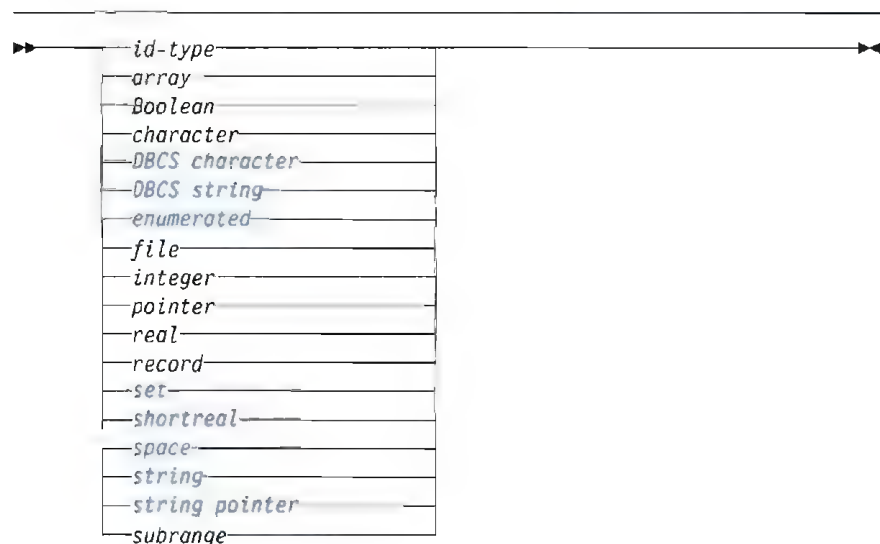


Figure 40. Syntax of a Data Type

Creating Your Own Data Types

Using the TYPE declaration, you can create your own data types. The identifiers for these new data types can then be used in type declarations for variables. For example, you can define data type COLOR as having the values RED, WHITE, and BLUE, and then define variable FLAG as being of type COLOR. A type identifier such as COLOR can be used anywhere a type definition is needed:

- In a variable declaration (VAR, **STATIC**, **DEF**, or **REF**)
- In a formal parameter declaration
- As a result type in a function header
- In a field declaration within a record definition
- In another TYPE declaration.

Type Compatibility

VS Pascal supports *strong typing* of data, which puts strict rules on what data types are considered to be the same. These rules for *type compatibility* require that you declare data carefully. The strong typing permits VS Pascal to check the validity of many operations at compile time, helping to produce reliable programs at run time.

Implicit Type Conversion

In general, VS Pascal does not perform implicit type conversions on data.

Figure 41 summarizes the only implicit conversions that VS Pascal performs.

A Value Of Type	Is Converted To Type	When
INTEGER	REAL or SHORTREAL	<ul style="list-style-type: none">One operand of a binary operation is an integer and the other is a real or shortreal.Assigned to a real or shortreal variable.Used in a floating-point divide operation (" / ").Passed to a parameter requiring a real or shortreal value. <p>Note: The type is converted only when non-VAR (pass-by value and pass-by-CONST) parameters are passed.</p>
SHORTREAL	REAL	<ul style="list-style-type: none">One operand of a binary operation is a shortreal and the other is a real.Assigned to a real variable.Passed to a parameter requiring a real value. <p>Note: The type is converted only when non-VAR (pass-by value and pass-by-CONST) parameters are passed.</p>
REAL	SHORTREAL	<ul style="list-style-type: none">Assigned to a shortreal variable.Passed to a parameter requiring a shortreal value. <p>Note: The type is converted only when non-VAR (pass-by value and pass-by-CONST) parameters are passed.</p>
STRING	SBCS Fixed String	<ul style="list-style-type: none">Assigned: The string is padded with blanks on the right if it is shorter than the array to which it is being assigned. Truncation produces a run-time error when checking is enabled.Passed to a formal parameter: The string is padded with blanks on the right if it is shorter than the array to which it is being passed. Truncation produces a run-time error when checking is enabled. <p>Note: The type is converted only when non-VAR (pass-by value and pass-by-CONST) parameters are passed.</p>

Figure 41 (Part 1 of 2). Implicit Type Conversions Performed by VS Pascal

A Value Of Type	Is Converted To Type	When
GSTRING	DBCS Fixed String	<ul style="list-style-type: none"> Assigned. The string is padded with blanks on the right if it is shorter than the array to which it is being assigned. Truncation produces a run-time error when checking is enabled. Passed to a formal parameter. The string is padded with blanks on the right if it is shorter than the array to which it is being passed. Truncation produces a run-time error when checking is enabled. <p>Note: The type is converted only when non-VAR (pass-by-value and pass-by-CONST) parameters are passed.</p>

Figure 41 (Part 2 of 2). Implicit Type Conversions Performed by VS Pascal

Same Data Types

Two variables are said to be of the *same type* when the declarations of the variables:

- Refer to the same type identifier
- Refer to different type identifiers that have been defined as equivalent by a type definition of the form:

```
TYPE T1 = T2;
```

- Appear in the same identifier list of a variable declaration or in the same formal parameter section of a routine heading. For example:

```
VAR V1, V2 : @INTEGER;
```

Variables declared with a type that is not a type identifier are said to have an *anonymous type*.

Compatible Data Types

Operations can be performed between two values that are of *compatible types*.

Two types are compatible when:

- Both types are the same.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types, and both are either packed or unpacked. In VS Pascal, matching packing of the sets is not required.
- Both are either SBCS or DBCS fixed strings with the same number of characters. In VS Pascal, the number of characters is not required to be the same.
- Both are either SBCS or DBCS variable-length strings.

Any object of type SET is compatible with the empty set. Any object that is a pointer type is compatible with the value NIL. String constants are compatible with character, fixed string, or variable-length string values, assuming that all length requirements are met.

Assignment Compatibility

A value can be assigned to a variable if the types are *assignment compatible*. An expression E is said to be assignment compatible with variable V if:

- The types are the same type, and neither type is a file nor contains a file.
- V is of type REAL and E is of type SHORTREAL.
- V is of type SHORTREAL and E is of type REAL.
- V is of type REAL or SHORTREAL and E is compatible with type INTEGER.
- V is a compatible subrange of E and the value to be assigned is within the allowable subrange of V.
- V and E have compatible set types and all members of E are permissible members of V.
- V is an SBCS or DBCS fixed string and E is an SBCS or DBCS string and the length of E is less than or equal to the length of V.
- V and E are both SBCS or DBCS variable-length strings and the length of E is less than or equal to the maximum length of V.

Figure 42 shows examples of type compatibility.

```

TYPE
    X      = ARRAY[ 1..10 ] OF
              INTEGER;
    Y      = X;
    DAYS   = (MON, TUE, WED, THU,
              FRI, SAT,  SUN);
    WEEKDAY = MON .. FRI;

VAR
    A : ARRAY[ 1..10 ] OF
          INTEGER;
    B : ARRAY[ 1..10 ] OF
          INTEGER;
    C,
    D : ARRAY[ 1..10 ] OF
          CHAR;
    E : X;
    F : X;
    G : Y;
    W1: DAYS;
    W2: WEEKDAY;

```

Variable:	Is compatible with:	Has the same type as:
A	A	A
B	B	B
C	C, D	C, D
D	D, C	D, C
E	E, F, G	E, F, G
F	F, E, G	F, E, G
G	G, E, F	G, E, F
W1	w1, w2	W1
W2	w2, w1	W2

Figure 42. Examples of Type Compatibility

Storage, Packing, and Alignment of Variables

For each variable declared with a particular type, VS Pascal allocates a specific amount of storage on a specific alignment boundary. *VS Pascal Application Programming Guide* describes implementation requirements and defaults.

When structured types are declared with the reserved word **PACKED**, data in the structure is not required to be aligned on the default boundaries. This may increase the run time of the program. Not all data types are affected by declaring them as **PACKED**.

VS Pascal Data Types

Figure 43 lists the data types by function. Descriptions of each data type, in alphabetical order, start on page 50.

Data Type	Associated Subtypes	Consists of	See Page
Enumerated scalar		A list of permitted values	57
Subrange scalar		A subset of consecutive values of a previously defined ordinal type	86
Predefined scalars	BOOLEAN	The values FALSE and TRUE	53
	CHAR	All the values of the EBCDIC character set	55
	GCHAR	All the values of the double-byte character set (DBCS)	60
	INTEGER	The subset of whole numbers from MININT (-2147483648) to MAXINT (2147483647)	64
	REAL	Double-precision floating-point data	67
	SHORTREAL	Single-precision floating-point data	79
ARRAY		A collection of homogeneous elements	51
RECORD		A collection of heterogeneous elements	69
SET		A collection of values taken from the same ordinal type	77
FILE		A one-dimensional sequence of components of the same type	59
Pointer		The address of a dynamic variable	66
Predefined structures	ALFA	An SBCS fixed string of 1 to 8 characters	50
	ALPHA	An SBCS fixed string of 1 to 16 characters	50
	GSTRING	A DBCS fixed string whose length varies up to a specified maximum	61
	STRING	An SBCS fixed string whose length varies up to a specified maximum	81
	TEXT	A FILE of CHAR	88
STRINGPTR		A predefined pointer to a variable of type STRING	84
SPACE		A storage allocation for variable-length data	81

Figure 43. Summary of VS Pascal Data Types

ALFA Data Type

The predefined type ALFA is defined as:

```
CONST
    ALFALEN = 8;

TYPE
    ALFA = PACKED
        ARRAY[1..ALFALEN] OF
            CHAR;
```

Figure 44 describes the operators and predefined functions that apply to the ALFA data type. See "Predefined Routines" on page 113 for further information about these predefined functions.

Operator or Function	Form	Description
=	Binary	Compares for equality
< > or ≠	Binary	Compares for inequality
<	Binary	Compares for left less than right
<=	Binary	Compares for left less than or equal to right
>=	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ADDR	Function	Returns the location in storage of a variable
HBOUND	Function	Returns ALFALEN
LBOUND	Function	Always returns the value 1
SIZEOF	Function	Returns the number of bytes required for a value of type ALFA, which is always 8
STR	Function	Converts type ALFA to type STRING

Figure 44. Operators and Predefined Functions for Type ALFA

ALPHA Data Type

The predefined type ALPHA is defined as:

```
CONST
    ALPHALEN = 16;

TYPE
    ALPHA = PACKED
        ARRAY[1..ALPHALEN] OF
            CHAR;
```

Figure 45 on page 51 describes the operators and predefined functions that apply to the ALPHA data type. See "Predefined Routines" on page 113 for further information about these predefined functions.

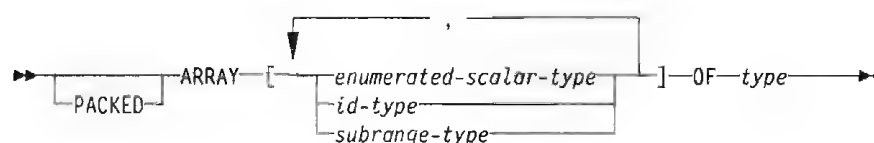
Operator or Function	Form	Description
<code>=</code>	Binary	Compares for equality
<code><</code> or <code><=</code>	Binary	Compares for inequality
<code><</code>	Binary	Compares for left less than right
<code><=</code>	Binary	Compares for left less than or equal to right
<code>>=</code>	Binary	Compares for left greater than or equal to right
<code>></code>	Binary	Compares for left greater than right
ADDR	Function	Returns the location in storage of a variable
HBOUND	Function	Returns ALPHALEN
IBOUND	Function	Always returns the value 1
SIZEOF	Function	Returns the number of bytes required for a value of type ALPHA, which is always 16
STR	Function	Converts type ALPHA to type STRING

Figure 46. Operators and Predetermined Functions for Type ALPHA

ARRAY Data Type

The ARRAY data type defines a list of homogeneous elements; each element is paired with one value of the index. The index can be any ordinal type. An element of the array is accessed through its subscript. To subscript a variable, you must specify an index. The number of elements in the array is the number of values potentially assumable by the index. Each element of the array is of the same type, which is called the *element type* of the array. Entire arrays can be assigned if they are of the same type.

Figure 46 shows the syntax of the ARRAY data type.



Where	Represents
<i>enumerated-scalar-type</i>	An enumerated scalar data type
<i>id-type</i>	An ordinal type name
<i>subrange-type</i>	A subrange data type
<i>type</i>	Any type

Figure 46. Syntax of the ARRAY Data Type

VS Pascal uses square brackets, "[" and "]", in the declaration of arrays. Because these symbols are not directly available on many I/O devices, the symbols "(" and ")" can be used as an alternative to square brackets.

VS Pascal aligns each element of the array, if necessary, to make each element fall on an appropriate boundary. A packed array will not observe the boundary

requirements of its elements. Standard Pascal does not allow elements of packed arrays to be passed by VAR to user-defined procedures.

An array defined with more than one index is said to be a multidimensional array. A multidimensional array is exactly equivalent to an array of arrays.

In short, an array definition of the form:

```
ARRAY[i,j,...] OF T
```

is an abbreviated form of

```
ARRAY[i] OF  
  ARRAY[j] OF  
    ... T
```

where *i* and *j* are ordinal type definitions.

Figure 47 shows examples of ARRAY declarations.

TYPE

```
MATRIX = ARRAY[1..10, 1..10] OF REAL;
```

```
MATRIX0 = ARRAY[1..10] OF      (* An alternative declaration *)  
          ARRAY[1..10] OF REAL; (* for MATRIX, above.          *)
```

```
ABLE = ARRAY[BOOLEAN] OF INTEGER;
```

```
COLOR = {RED, YELLOW, BLUE};
```

```
INTENSITY = PACKED ARRAY [COLOR] OF REAL;
```

```
ALFA = PACKED ARRAY [1..ALFALEN] OF CHAR;
```

Figure 47 Examples of ARRAY Declarations

Figure 48 describes the predefined routines that apply to the ARRAY data type. See "Predefined Routines" on page 113 for further information about these predefined routines.

Routine	Form	Description
ADDR	Function	Returns the location in storage of an array variable
HBOUND	Function	Determines the upper bound of a dimension in an array
LBOUND	Function	Determines the lower bound of a dimension in an array
PACK	Procedure	Copies an array starting at a given point to a packed array
SIZEOF	Function	Returns the number of bytes required for an array
UNPACK	Procedure	Copies a packed array to an array starting at a given point

Figure 48. Predefined Routines for Type ARRAY

BOOLEAN Data Type

The predefined data type BOOLEAN is defined as an ordinal type whose values are FALSE and TRUE, as though declared with the following type declaration:

```
TYPE  
  BOOLEAN=(FALSE,TRUE);
```

Variables of this type occupy 1 byte of storage and are aligned on a byte boundary.

Figure 49 describes the operators and predefined functions that apply to the BOOLEAN data type. See “Predefined Routines” on page 113 for further information about these predefined functions.

Operator or Function	Form	Description
NOT or \neg	Unary	Returns TRUE if the operand is FALSE; otherwise, it returns FALSE
AND or &	Binary	Returns TRUE if both operands are TRUE
OR or	Binary	Returns TRUE if either operand is TRUE
\oplus or XOR or &&	Binary	Returns TRUE if only one of the operands is TRUE
=	Binary	Compares for equality
< > or \neq	Binary	Compares for inequality
<	Binary	Compares for left less than right
< =	Binary	Compares for left less than or equal to right
> =	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ADDR	Function	Returns the location in storage of a Boolean variable
HIGHEST	Function	Returns TRUE by definition
LOWEST	Function	Returns FALSE by definition
MAX	Function	Returns FALSE if all operands are FALSE; otherwise, it returns TRUE
MIN	Function	Returns TRUE if all operands are TRUE; otherwise, it returns FALSE
ORD	Function	Returns 0 if an expression is FALSE, and 1 if an expression is TRUE
SIZEOF	Function	Returns the number of bytes required for a value of type BOOLEAN, which is always 1

Figure 49. Operators and Predefined Functions for Type BOOLEAN

The relational operators form valid Boolean functions as shown in Figure 50.

Name	Operator	Result
= (Equivalence)	FALSE = FALSE	TRUE
	FALSE = TRUE	FALSE
	TRUE = FALSE	FALSE
	TRUE = TRUE	TRUE
< > or \neg = (Exclusive OR)	FALSE < > FALSE	FALSE
	FALSE < > TRUE	TRUE
	TRUE < > FALSE	TRUE
	TRUE < > TRUE	FALSE
<	FALSE < FALSE	FALSE
	FALSE < TRUE	TRUE
	TRUE < FALSE	FALSE
	TRUE < TRUE	FALSE
< = (Implication)	FALSE < = FALSE	TRUE
	FALSE < = TRUE	TRUE
	TRUE < = FALSE	FALSE
	TRUE < = TRUE	TRUE
> =	FALSE > = FALSE	TRUE
	FALSE > = TRUE	FALSE
	TRUE > = FALSE	TRUE
	TRUE > = TRUE	TRUE
>	FALSE > FALSE	FALSE
	FALSE > TRUE	FALSE
	TRUE > FALSE	TRUE
	TRUE > TRUE	FALSE
AND or &	FALSE&FALSE	FALSE
	FALSE&TRUE	FALSE
	TRUE&FALSE	FALSE
	TRUE&TRUE	TRUE
OR or	FALSE FALSE	FALSE
	FALSE TRUE	TRUE
	TRUE FALSE	TRUE
	TRUE TRUE	TRUE

Figure 50 (Part 1 of 2). Relational Operators on Type BOOLEAN

Name	Operator	Result
or XOR or &&	FALSE > < FALSE	FALSE
	FALSE > < TRUE	TRUE
	TRUE > < FALSE	TRUE
	TRUE > < TRUE	FALSE

Figure 50 (Part 2 of 2). Relational Operators on Type BOOLEAN

For information on the evaluation of Boolean expressions, see "BOOLEAN Expressions" on page 199.

CHAR Data Type

The predefined data type CHAR is an ordinal whose values represent the EBCDIC character set. Variables of this type occupy 1 byte of storage and are aligned on a byte boundary.

A single-character string constant will be regarded as a CHAR constant if the context so dictates. For example, this assignment statement:

```
VAR
  C: CHAR;
BEGIN
  .
  .
  C := 'A';
  .
  .
END;
```

sets variable C to the EBCDIC code for the letter A.

Figure 51 describes the operators and predefined functions that apply to the CHAR data type. See "Predefined Routines" on page 113 for further information about these predefined functions.

Operator or Function	Form	Description
=	Binary	Compares for equality
< > or <= >=	Binary	Compares for inequality
<	Binary	Compares for left less than right
<=	Binary	Compares for left less than or equal to right
>=	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ADDR	Function	Returns the location in storage of a variable
HIGHEST	Function	Returns CHR(255)

Figure 51 (Part 1 of 2). Operators and Predefined Functions for Type CHAR

Operator or Function	Form	Description
LOWEST	Function	Returns CHR(0)
MAX	Function	Returns the maximum value of one or more operands
MIN	Function	Returns the minimum value of one or more operands
ORD	Function	Converts a character to an integer based on the ordering sequence of the underlying character set
PRED	Function	Returns the preceding character in the ordering sequence of the underlying character set
SIZEOF	Function	Returns the number of bytes required for a value of type CHAR, which is always 1
STR	Function	Converts either a CHAR or an SBCS fixed string to a STRING
SUCC	Function	Returns the succeeding character in the ordering sequence of the underlying character set

Figure 51 (Part 2 of 2). Operators and Predefined Functions for Type CHAR

See Figure 90 on page 84 to see how binary operators are applied to SBCS characters. See Figure 91 on page 84 to see how to convert SBCS strings on assignment.

DBCS Fixed String Data Type

A DBCS fixed string is defined as a PACKED ARRAY [1..n] OF GCHAR, where *n* is a positive integer constant. All operations on DBCS data are handled in a byte-oriented manner.

Figure 52 describes the operators and predefined routines that apply to the DBCS fixed string data type. See "Predefined Routines" on page 113 for further information about these predefined routines.

Operator or Routine	Form	Description
=	Binary	Compares for equality ¹
< > or \neq	Binary	Compares for inequality ¹
<	Binary	Compares for left less than right ^{1,2}
<=	Binary	Compares for left less than or equal to right ^{1,2}
>=	Binary	Compares for left greater than or equal to right ^{1,2}
>	Binary	Compares for left greater than right ^{1,2}
ADDR	Function	Returns the location in storage of a DBCS fixed string
GSTR	Function	Converts a DBCS fixed string to a GSTRING
HBOUND	Function	Determines the upper bound of a DBCS fixed string

Figure 52 (Part 1 of 2). Operators and Routines for the DBCS Fixed String Data Type

Operator or Routine	Form	Description
LBOUND	Function	Determines the lower bound of a DBCS fixed string
PACK	Procedure	Copies an array starting at a given point to a packed array
SIZEOF	Function	Returns the number of bytes required for a DBCS fixed string
UNPACK	Procedure	Copies a packed array to an array starting at a given point

Figure 52 (Part 2 of 2). Operators and Routines for the DBCS Fixed String Data Type

Notes to Figure 52:

1. If two strings being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.
2. Relative magnitude of two DBCS fixed strings is based upon the binary value of DBCS codes.

See Figure 62 on page 63 to see how binary operators are applied to DBCS fixed strings. See Figure 63 on page 64 to see how to convert DBCS strings on assignment.

Enumerated Scalar Data Type

An enumerated scalar is formed by listing each value permitted for a particular type of variable. This allows a meaningful name to be associated with each value. Figure 53 shows the syntax of the enumerated scalar data type.



Where Represents

id An identifier that is treated as a self-defining constant

Figure 53. Syntax of the Enumerated Scalar Data Type

An enumerated scalar type definition declares the identifiers in the enumeration list as constants of the same type as the enumerated scalar that is being defined. The lexical scope of the newly defined constants is the same as that of any other identifier declared explicitly at the same lexical level.

These constants are ordered such that the first value is less than the second, the second less than the third, and so forth. For example, in the first of the following examples, `MON < TUE < WED < ... < SUN`; there is no value less than the first or greater than the last.

Figure 54 shows examples of enumerated scalar data types.

```
TYPE
  DAYS      = (MON, TUE, WED, THU,
               FRI, SAT, SUN);

  MONTHS    = (JAN, FEB, MAR, APR,
               MAY, JUN, JUL, AUG,
               SEP, OCT, NOV, DEC);

VAR
  SHAPE     : (TRIANGLE, RECTANGLE,
               SQUARE, CIRCLE);

  REC       : RECORD
    SUIT: (SPADE, HEART,
           DIAMOND, CLUB);
    WEEK: DAYS
  END;

  MONTH     : MONTHS;
```

Figure 54. Examples of Enumerated Scalar Data Types

Note: Two enumerated scalar type definitions must not have any elements of the same name in the same lexical scope.

Figure 55 describes the predefined functions that apply to the enumerated scalar data type. See “Predefined Routines” on page 113 for further information about these predefined functions.

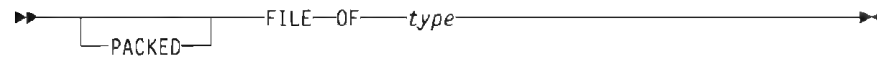
Function	Description
ADDR	Returns the location in storage of an enumerated scalar type
HIGHEST	Returns the maximum value of an enumerated scalar
LOWEST	Returns the minimum value of an enumerated scalar
MAX	Returns the maximum value of one or more enumerated scalar expressions
MIN	Returns the minimum value of one or more enumerated scalar expressions
ORD	Converts an ordinal expression to an integer
PRED	Returns the predecessor of an ordinal expression
SIZEOF	Returns the number of bytes required for a value of an enumerated scalar type
SUCC	Returns the successor of an ordinal expression

Figure 55. Predefined Functions for Enumerated Scalar Data Type

FILE Data Type

Input and output in VS Pascal is usually done through a file. A *file* is a structure consisting of a sequence of components in which each component is of the same type. Variables of this type reference the components with pointers called *file pointers*. A file pointer can be thought of as a pointer into an input/output buffer.

Figure 56 shows the syntax of the FILE data type.



Where	Represents
<i>type</i>	Any data type that does not contain a file

Figure 56. Syntax of the FILE Data Type

Note to Figure 56: A file declared like this is called a *record file*. You can declare a file to be PACKED, but the declaration has no effect on the file's storage requirements.

The association of a file variable to an actual file of the system is implementation-dependent and will not be described in this manual. See the *VS Pascal Application Programming Guide* for further information.

Figure 57 shows examples of FILE declarations

```
TYPE
  TEXT = FILE OF CHAR;
  LINE = FILE OF
    PACKED ARRAY[1..80] OF CHAR;
  PFILE = FILE OF
    RECORD
      NAME : PACKED
        ARRAY[1..25] OF CHAR;
      PERSON_NO : INTEGER;
      DATE_EMPLOYED : DATE;
      WEEKLY_SALARY : INTEGER;
    END;
```

Figure 57. Examples of FILE Declarations

Restrictions:

- A file cannot be contained within a file.
- A file or a structure containing a file cannot be used in an assignment statement.
- A file or a structure containing a file cannot be passed by value; it must be passed by VAR or CONST.
- A type containing a file cannot be used as the type name of a structured constant

You access file variables using predefined functions.

Figure 58 describes the routines that apply to record file variables. See “Predefined Routines” on page 113 for further information about these functions.

Routine	Form	Description
ADDR	Function	Returns the location in storage of a variable
CLOSE	Procedure	Closes a file
EOF	Function	Tests a file for end-of-file condition
GET	Procedure	Reads the current element of a file and advances the file pointer to the next element in the input file
PDSIN	Procedure	Opens a file for input, specifying the open options and the member name of the PDS
PDSOUT	Procedure	Opens a file for output, specifying the open options and the member name of the PDS
PUT	Procedure	Writes the file buffer to a file and advances the file pointer to the next element in the output file
READ	Procedure	Reads from a file into a variable
RESET	Procedure	Opens a file for input with the open options
REWRITE	Procedure	Opens a file for output with the open options
SEEK	Procedure	Positions a file to a specified component
SIZEOF	Function	Returns the number of bytes required for a value of type FILE
UPDATE	Procedure	Opens a file for both input and output with the open options
WRITE	Procedure	Writes the value of an expression to a file

Figure 58. Routines for Record File Variables

GCHAR Data Type

The predefined data type GCHAR is a scalar that represents one double-byte character set (DBCS) character. GCHAR variables occupy 2 bytes of memory and are aligned on a halfword boundary.

Because values of GCHAR are not mapped on consecutive integers, GCHAR is not an ordinal type. Therefore, the GCHAR type cannot be used in certain circumstances. For example:

- In subranges or sets of GCHAR
- As an array index type
- As a CASE selector
- As the type name of an ordinal conversion routine
- As the type of variable in a FOR loop index
- As a type of variant selector
- In the predefined functions SUCC, PRED, ORD, HIGHEST, and LOWEST.

A DBCS string constant will be regarded as a GCHAR constant if the context so dictates. For example, this assignment statement:

```
VAR
  C: GCHAR;
BEGIN
  .
  .
  C := '<.A>'G;
  .
  .
END;
```

sets variable C to the DBCS code for the letter A.

Figure 59 describes the operators and predefined functions that apply to the GCHAR data type. See "Predefined Routines" on page 113 for further information about these predefined functions.

Operator or Function	Form	Description
=	Binary	Compares for equality
< > or \neq	Binary	Compares for inequality
<	Binary	Compares for left less than right
< =	Binary	Compares for left less than or equal to right
> =	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ADDR	Function	Returns the location in storage of a GCHAR variable
GSTR	Function	Converts a GCHAR to a GSTRING
MAX	Function	Returns the maximum value of one or more operands
MIN	Function	Returns the minimum value of one or more operands
SIZEOF	Function	Returns the number of bytes required for a value of type GCHAR, which is always 2

Figure 59. Operators and Predefined Functions for Type GCHAR

See Figure 62 on page 63 to see how binary operators are applied to DBCS characters. See Figure 63 on page 64 to see how to convert DBCS strings on assignment.

GSTRING Data Type

The predefined data type GSTRING is defined as:

```
TYPE
  GSTRING = PACKED ARRAY[1..N] OF GCHAR;
```

A variable declared as GSTRING(n) occupies $2 \cdot n$ bytes for data, plus 2 bytes for a length field. The maximum length is 16,382 DBCS characters.

Figure 60 shows the syntax of the GSTRING data type.

➤ GSTRING (*constant-expr*) ➤

Where	Represents
<i>constant-expr</i>	Any constant expression

Figure 60. Syntax of the GSTRING Data Type

The length of a GSTRING variable varies at run time from 0 up to its declared maximum, which is fixed at compile time. In this example,

```
A : GSTRING(10);
```

variable A can, at run time, be any length from 1 to 10 characters. Variable A's maximum length of 10 characters was fixed at compile time.

The length of a GSTRING variable is managed implicitly by the operators and functions that apply to strings. For example, to obtain the actual length of a GSTRING at run time, use the LENGTH function (see "LENGTH Function" on page 131). To obtain the maximum length at run time, use the MAXLENGTH function (see "MAXLENGTH Function" on page 137).

Figure 61 describes the operators and predefined routines that apply to the GSTRING data type. See "Predefined Routines" on page 113 for further information about these routines.

Operator or Routine	Form	Description
=	Binary	Compares for equality ¹
< > or \neq	Binary	Compares for inequality ¹
<	Binary	Compares for left less than right ^{1,2}
<=	Binary	Compares for left less than or equal to right ^{1,2}
>=	Binary	Compares for left greater than or equal to right ^{1,2}
>	Binary	Compares for left greater than right ^{1,2}
+ or	Binary	Concatenates the operands
ADDR	Function	Returns the location in storage of a variable
COMPRESS	Function	Returns a DBCS string with all occurrences of multiple blanks replaced by a single blank
DELETE	Function	Returns a DBCS string with a portion removed
GSTR	Function	Converts a GCHAR or a DBCS fixed string to a GSTRING
GTOSTR	Function	Converts a GSTRING to a STRING, adding shift-out and shift-in characters to the STRING
INDEX	Function	Locates the first occurrence of a DBCS string in another DBCS string
LENGTH	Function	Returns the length of a DBCS string

Figure 61 (Part 1 of 2). Operators and Predefined Routines for Type GSTRING

Operator or Routine	Form	Description
LPAD	Procedure	Pads or truncates a DBCS string on the left
LTRIM	Function	Returns a DBCS string with leading blanks removed
MAXLENGTH	Function	Returns the declared length of a DBCS string
RINDEX	Function	Locates the last occurrence of a DBCS string in another DBCS string
RPAD	Procedure	Pads or truncates a DBCS string on the right
SIZEOF	Function	Returns the number of bytes required for a value of type GSTRING
SUBSTR	Function	Returns a specified portion of a DBCS string
TRIM	Function	Returns a DBCS string with trailing blanks removed

Figure 61 (Part 2 of 2) Operators and Predefined Routines for Type GSTRING

Notes to Figure 61:

1. If two GSTRINGs being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.
2. Relative magnitude of two GSTRINGs is based upon the binary value of DBCS codes.

Figure 62 shows how binary operators are applied to DBCS characters, DBCS fixed strings, and DBCS strings.

Left Operand	Right Operand	Result
GCHAR	GCHAR	Allowed
	DBCS Fixed String	Use GSTR on both operands
	GSTRING	Use GSTR on the GCHAR
DBCS Fixed String	GCHAR	Use GSTR on both operands
	DBCS Fixed String	Allowed if the types are compatible
	GSTRING	Use GSTR on the array
GSTRING	GCHAR	Use GSTR on the GCHAR
	DBCS Fixed String	Use GSTR on the array
	GSTRING	Allowed

Figure 62. How to Apply Binary Operators to DBCS Characters, DBCS Fixed Strings, and DBCS Strings

Figure 63 shows how to convert DBCS strings on assignment.

Target Variable	Source Expression	Result
GCHAR	GCHAR	Allowed
	DBCS Fixed String	Index the array to obtain a GCHAR
	GSTRING	Index the string to obtain a GCHAR
DBCS Fixed String	GCHAR	Not permitted
	DBCS Fixed String	Allowed if the types are compatible
	GSTRING	Allowed if truncation is required; an error results
GSTRING	GCHAR	Use GSTR to convert GCHAR to a GSTRING
	DBCS Fixed String	Use GSTR to convert array to a GSTRING; if truncation is required, an error results
	GSTRING	Allowed; if truncation is required, an error results

Figure 63 How to Convert DBCS Strings on Assignment

INTEGER Data Type

The predefined data type INTEGER represents the subset of whole numbers as defined below:

```
TYPE
    INTEGER =MININT ..MAXINT;
```

where MININT is a predefined integer constant whose value is -2147483648 and MAXINT is a predefined integer constant whose value is 2147483647. The predefined type INTEGER represents 32-bit values in 2's complement notation. Variables of this type occupy 4 bytes of storage and are aligned on a fullword boundary. See "Subrange Data Type" on page 86 for a discussion of integer subranges.

Figure 64 describes the operators and predefined functions that apply to the INTEGER data type. See "Predefined Routines" on page 113 for further information about these functions.

Operator or Function	Form	Description
+	Unary	Returns the unchanged result of the operand
+	Binary	Forms the sum of the operands
-	Unary	Negates the operand

Figure 64 (Part 1 of 2). Operators and Predefined Functions for Type INTEGER

Operator or Function	Form	Description
-	Binary	Forms the difference of the operands
*	Binary	Forms the product of the operands
/	Binary	Converts the operands to reals and produces the real quotient
DIV	Binary	Forms the integer quotient of the operands
MOD	Binary	Forms the integer modulus of the operands (same as remainder if the arguments are positive)
=	Binary	Compares for equality
< > or !=	Binary	Compares for inequality
<	Binary	Compares for left less than right
<=	Binary	Compares for left less than or equal to right
>=	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
NOT or ~	Unary	Returns one's complement of the operand
OR or	Binary	Returns the logical or of the operands
AND or &	Binary	Returns the logical and of the operands
^ or XOR or &&	Binary	Returns the logical exclusive or of the operands
<<	Binary	Returns the left operand value shifted left by the right operand value
>>	Binary	Returns the left operand value shifted right by the right operand value
ABS	Function	Returns the absolute value of a number
ADDR	Function	Returns the location in storage of a variable
CHR	Function	Returns a character corresponding to an integer
FLOAT	Function	Converts an integer to a real number
HIGHEST	Function	Returns MAXINT
LOWEST	Function	Returns MININT
MAX	Function	Returns the maximum value of one or more operands
MIN	Function	Returns the minimum value of one or more operands
ODD	Function	Returns TRUE if an expression is odd, returns FALSE otherwise
ORD	Function	Returns the value of a passed expression
PRED	Function	Subtracts 1 from an integer
SIZEOF	Function	Returns the number of bytes required for an integer
SQR	Function	Returns the square of a number
SUCC	Function	Adds 1 to an integer

Figure 64 (Part 2 of 2). Operators and Predefined Functions for Type INTEGER

Notes to Figure 64 on page 64:

- The operations of DIV and MOD are defined as:

$A \text{ DIV } B = \text{TRUNC}(A/B), B \neq 0$
 $A \text{ MOD } B = A - B * (A \text{ DIV } B), A \geq 0, B > 0$
 $A \text{ MOD } B = B - \text{ABS}(A) \text{ MOD } B, A < 0, B > 0$

If $B = 0$ when doing a DIV operation, or if $B < -0$ when doing a MOD operation you will get a run-time error message.

- The following operators perform logical operations:

<<	shift left logical
>>	shift right logical
~	one's complement
or OR	logical inclusive or
& or AND	logical and
>> or XOR or &&	logical exclusive or

The operands are treated as unsigned strings of binary digits. See "Logical Expressions" on page 202 for more details on logical expressions.

Caution: Intermediate result overflow may or may not be caught when performing certain arithmetic operations on integers.

Pointer Data Type

Using the predefined procedure NEW, you can allocate storage for dynamic variables at run time. These dynamic variables are allocated in an area of storage called a *heap*. *Pointer variables* keep track of these dynamic variables by maintaining their addresses in storage. Figure 65 shows the syntax of the pointer data type.



Where Represents
id type Any type

Figure 65. Syntax of the Pointer Data Type

The pointer declaration must specify the data type of the dynamic variable to be created. Figure 66 provides an example of pointer declarations.

```
TYPE
  PTR = @ELEMENT;
  ELEMENT = RECORD
    PARENT : PTR;
    CHILD  : PTR;
    SIBLING: PTR;
  END;
```

Figure 66. Example of Pointer Declarations

Note to Figure 66: The identifier ELEMENT is used before it is declared. Although referencing an identifier before it is declared is generally not permitted in VS Pascal, one exception is a type identifier used as the base type in a pointer declaration.

Figure 67 describes the operators and predefined routines that apply to the pointer data type. See “Predefined Routines” on page 113 for further information about these routines.

Operator or Routine	Form	Description
<code>=</code>	Binary	Compares for equality
<code>< > or \rightarrow =</code>	Binary	Compares for inequality
<code>ADDR</code>	Function	Returns the location in storage of a variable
<code>DISPOSE</code>	Procedure	Deallocates a dynamic variable pointed to by a pointer
<code>DISPOSEHEAP</code>	Procedure	Deallocates a heap
<code>MARK</code>	Procedure	Creates a subheap in the current heap
<code>NEW</code>	Procedure	Allocates a dynamic variable in the current heap and sets the pointer to point to the dynamic variable
<code>NEWHEAP</code>	Procedure	Allocates a new heap
<code>ORD</code>	Function	Converts a pointer to an integer equal to the address of the dynamic variable pointed to by the pointer (see note)
<code>QUERYHEAP</code>	Procedure	Sets the pointer to the current heap
<code>RELEASE</code>	Procedure	Frees all subheaps created after a specified subheap in the heap containing the specified subheap
<code>SIZEOF</code>	Function	Returns the number of bytes required for a pointer
<code>USEHEAP</code>	Procedure	Changes the current heap

Figure 67. Operators and Predefined Routines for the Pointer Data Type

Note to Figure 67: There is no function in VS Pascal to convert an integer into a pointer.

REAL Data Type

The predefined data type REAL represents floating-point data. Variables of this type occupy 8 bytes of storage and are aligned on a doubleword boundary. All real arithmetic is done using double-precision floating-point instructions.

`MAXREAL` is a predefined constant whose value is the largest floating-point number representable on the machine. `MINREAL` is a predefined constant whose value is the smallest positive non-zero floating-point number representable on the machine. `EPSREAL` is a predefined constant whose value is the smallest number representable on the machine such that $1.0 + \text{EPSREAL} > 1.0$.

Because the REAL type is not ordinal, it cannot be used in certain circumstances. For example:

- In subranges or sets of REAL
- As an array index type
- As a CASE selector
- As the type of variable in a FOR loop index
- As a type of variant selector
- In the predefined functions SUCC, PRED, ORD, HIGHEST, and LOWEST.

Figure 68 describes the operators and predefined functions that apply to the REAL data type. See "Predefined Routines" on page 113 for further information about these functions.

Operator or Function	Form	Description
+	Unary	Returns the unchanged result of the operand
+	Binary	Forms the sum of the operands
-	Unary	Negates the operand
-	Binary	Forms the difference of the operands
*	Binary	Forms the product of the operands
/	Binary	Forms the floating-point quotient of the operands
=	Binary	Compares for equality
< > or <=	Binary	Compares for inequality
<	Binary	Compares for left less than right
<=	Binary	Compares for left less than or equal to right
>=	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ABS	Function	Returns the absolute value of a number
ADDR	Function	Returns the location in storage of a variable
ARCTAN	Function	Returns the trigonometric arctangent (in radians) of a real argument
COS	Function	Returns the trigonometric cosine of a real argument (in radians)
EXP	Function	Returns the value of the natural log base raised to the power of a real argument
LN	Function	Returns the natural logarithm of a real argument
MAX	Function	Returns the maximum value of one or more operands
MIN	Function	Returns the minimum value of one or more operands
ROUND	Function	Returns a value rounded to an integer
SIN	Function	Returns the trigonometric sine of a real argument (in radians)

Figure 68 (Part 1 of 2). Operators and Predefined Functions for Type REAL

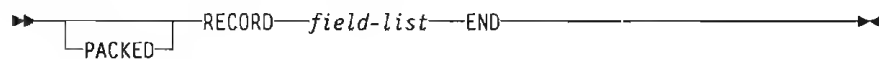
Operator or Function	Form	Description
SIZEOF	Function	Returns the number of bytes required for type REAL, which is always 8
SQR	Function	Returns the square of a number
SQRT	Function	Returns the square root of a real argument
TRUNC	Function	Returns a value truncated to an integer

Figure 68 (Part 2 of 2). Operators and Predefined Functions for Type REAL

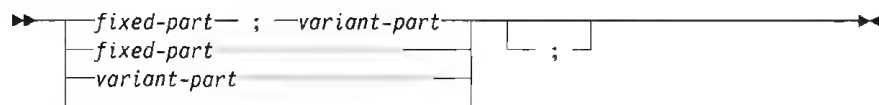
RECORD Data Type

A *record* is a data structure composed of heterogeneous components; each element can be a different type. Components of a record are called *fields*. Figure 69 shows the syntax of the RECORD data type.

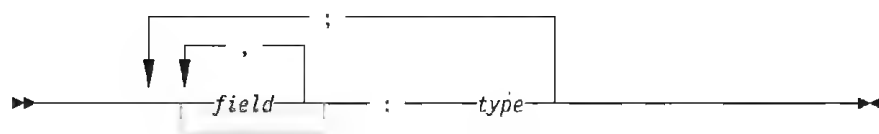
Record type



Field-list



Fixed-part



Variant-part

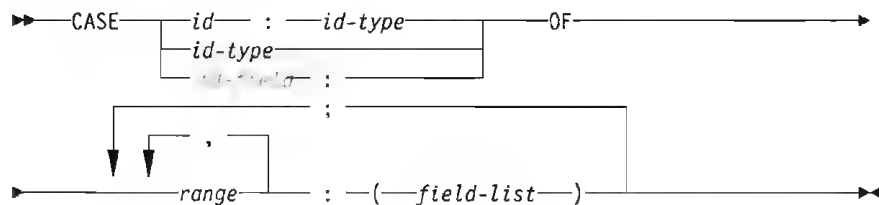


Figure 69 (Part 1 of 2). Syntax of the RECORD Data Type

Field**Range**

Where	Represents
<i>constant</i>	Any constant
<i>constant-expr</i>	Any constant expression
<i>type</i>	Any type
<i>id</i>	An identifier name
<i>id-type</i>	A type name
<i>id-field</i>	A field name

Figure 69 (Part 2 of 2). Syntax of the RECORD Data Type

Naming a Field

A field is referred to by the name it has been assigned. The scope in which a field name is valid is within the record in which the field is declared. Every field name within a record must be unique, even if that name appears in a variant part.

A field of a record need not be named; the field identifier can be missing. In such a case, the field serves only as padding and cannot be referenced.

Figure 70 shows examples of simple RECORD declarations.

```
TYPE
  REC = RECORD
    A,
    B : INTEGER;
    : CHAR;      (*unnamed*)
    C : CHAR;
  END;

  DATE = RECORD
    DAY   : 1..31;
    MONTH : 1..12;
    YEAR  : 1900..2100
  END;

  PERSON = RECORD
    LAST_NAME,
    FIRST_NAME : ALFA;
    MIDDLE_INITIAL : CHAR;
    AGE : 0..99;
    EMPLOYED : BOOLEAN;
  END;
```

Figure 70. Examples of Simple RECORD Declarations

Fixed Part

The fixed part of a record is a series of fields that exists in every variable declared to be of that record type. The fixed part, if present, is always before the variant part.

Variant Selector

The *variant selector* follows the reserved word **CASE** in the variant part of a record. This is an ordinal type that indicates which variant of the record is active.

When the variant selector is followed by a colon, a new field called the *tag field* is defined. For example,

```
CASE I: INTEGER OF
```

results in **I** being a tag field of type **INTEGER**.

If the type identifier is missing, the tag field name must be one previously defined within the record. This allows you to place the tag field anywhere in the fixed part of the record. For example,

```
CASE I: OF
```

means that **I** is the tag field, and it must have been declared in the fixed part. The type of **I** is as given in the field definition of **I**.

The variant part of a record need not have a tag field at all. In this case, only a type identifier is specified in the case construct. For example,

```
CASE INTEGER OF
```

means no tag field is present; the variants are denoted by integer values in the variant declaration. You must still refer to the variant fields by their names, but it is your responsibility to keep track of which variant is "active" (that is, contains valid data) at run time.

Variant Part

You use the variant part of a record to define an alternative structure in the record. The record structure adopts one of the variants at a time.

All variant tags must be assignment compatible with the variant selector type. Also, in Standard Pascal, all possible values of a variant selector must correspond to a variant. With VS Pascal, you can omit those tag constants that will not be used

Figure 71 on page 72 illustrates how to declare a record with a tag field.

```

TYPE

  SHAPE = (TRIANGLE, RECTANGLE,
           SQUARE,  CIRCLE);

  COORDINATES =
      (*fixed part: *)
  RECORD
    X,Y   : REAL;
    AREA  : REAL;
    CASE S : SHAPE OF
      (*variant part:*)
    TRIANGLE:
      (SIDE : REAL;
       BASE : REAL);

    RECTANGLE:
      (SIDEA,SIDEB : REAL);

    SQUARE:
      (EDGE : REAL);

    CIRCLE:
      (RADIUS : REAL);
  END;

```

Figure 71. Example of a Record Declaration with a Tag Field

In Figure 71, the record defined as COORDINATES contains a variant part. The tag field is S, its type is SHAPE, and its value (whether TRIANGLE, RECTANGLE, SQUARE, or CIRCLE) indicates which variant is in effect. The fields SIDE, SIDEA, EDGE, and RADIUS will all occupy the same offset within the record. Figure 72 shows how the record will look in storage.

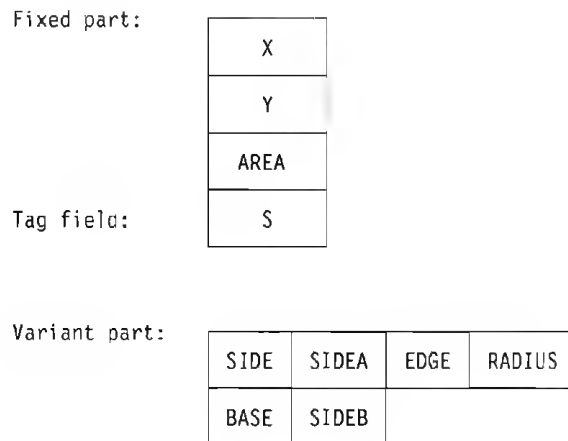


Figure 72. Storage of a Record with a Tag Field

Each column in the variant represents one alternative for the variant.

If you prefer the tag field to be the first field instead of the fourth, define it as shown in Figure 73 on page 73

```

COORDINATES =
RECORD
  S      : SHAPE;
  X,Y    : REAL;
  AREA   : REAL;
  CASE S : OF
    (*variant part:*)
  TRIANGLE:
    (SIDE : REAL;
     BASE : REAL);

  RECTANGLE:
    (SIDEA,SIDE B : REAL);

  SQUARE:
    (EDGE : REAL);

  CIRCLE:
    (RADIUS : REAL);
END;

```

Figure 73. Example of a Record Declaration with a Back Reference Tag Field

Figure 74 shows how the record will look in storage.



Figure 74 Storage of a Record with a Back Reference Tag Field

If you prefer the tag field to be absent altogether, define the record as shown in Figure 75 on page 74.

```

COORDINATES =
RECORD
  X,Y    : REAL;
  AREA   : REAL;
  CASE SHAPE OF
    (*variant part:*)
  TRIANGLE:
    (SIDE : REAL;
     BASE : REAL);

  RECTANGLE:
    (SIDEA.SIDE B : REAL);

  SQUARE:
    (EDGE : REAL);

  CIRCLE:
    (RADIUS : REAL);
END;

```

Figure 75 Example of a Record Variant with No Tag Field

Figure 76 shows how the record will look in storage.

Fixed part:

X
Y
AREA

Variant part:

SIDE	SIDEA	EDGE	RADIUS
BASE	SIDE B		

Figure 76. Storage of a Record Variant with No Tag Field

Packed Records

The fields in a record are normally assigned offsets sequentially and padded where necessary for boundary alignment. In packed records, however, no such padding is done, and fields are aligned on a byte boundary. This might save storage within the record, but might also degrade performance of the program.

Standard Pascal does not allow elements of packed records to be passed by VAR to user-defined procedures.

Offset Qualification of Fields

With VS Pascal you can force the fields of a record to begin at a specified byte offset in the record. A field name can be followed by an integer constant expression enclosed in parentheses. This expression represents the byte offset within the record where the field begins. All fields so specified must be in consecutive order according to offsets. If the offset is not specified, the field will be assigned the next offset required for boundary alignment. If an offset specification attempts to assign an incorrect boundary for a field and the record is not packed, a compile-time error will be issued.

For example, a large control block of 100 bytes is needed in which four fields at various offsets must be referenced. The fields of the control block, and how the control block might be represented in VS Pascal, are shown in Figure 77

Byte Displacement	Information
0	Field A (integer)
36	Field B (8 characters)
80	Field C (4 flags)
92	Field D (integer)


```
TYPE
  FLAGS = SET OF (F1,F2,F3,F4);
  PADDING = PACKED ARRAY[1..4] OF CHAR;
  CB = PACKED RECORD
    A      : INTEGER;
    B(36)  : ALFA;
    C(80)  : FLAGS;
    D(92)  : INTEGER;
           : PADDING
  END;
VAR
  BLOCK : CB;
```

Figure 77. Example of a Record with Offset Qualified Fields

You cannot use an offset qualifier on the variant part tag field. If you want the tag field to be at a certain offset, make the tag field a backward reference, make the last identifier of the fixed part have the same name as the tag field, and put the offset qualifier on this last identifier. Figure 78 on page 76 illustrates this point.

```

TYPE
  TAG =
    PACKED RECORD
      A : BOOLEAN;
      B(3) : BOOLEAN;

      CASE B : OF
        TRUE:
          (I : INTEGER)

        FALSE:
          (P : 0INTEGER)
      END;
VAR
  BLOCK : TAG;

```

Figure 78 Example of an Offset Qualifier on a Tag Field

Figure 79 describes the predefined functions that apply to the RECORD data type. See “Predefined Routines” on page 113 for further information about these functions.

Function	Form	Description
ADDR	Function	Returns the location in storage of a record variable
SIZEOF	Function	Returns the number of bytes required for a value of type RECORD

Figure 79. Predefined Functions for Type RECORD

SBCS Fixed String Data Type

An SBCS fixed string is defined as a PACKED ARRAY [1..n] OF CHAR. In Standard Pascal, *n* must be greater than 1, and any two strings that are compared or assigned must have the same length. In VS Pascal, *n* can equal 1, and the strings need not be the same length.

Double-byte character set (DBCS) data is allowed in an SBCS fixed string when it is surrounded by shift-in and shift-out characters. The GRAPHIC compile-time option has no effect on this support. All operations on DBCS data are handled in a byte-oriented manner.

Figure 80 on page 77 describes the operators and predefined routines that apply to the SBCS fixed string data type. See “Predefined Routines” on page 113 for further information about these predefined routines.

Operator or Routine	Form	Description
=	Binary	Compares for equality ¹
< > or <= >=	Binary	Compares for inequality ¹
<	Binary	Compares for left less than right ^{1,2}
<=	Binary	Compares for left less than or equal to right ^{1,2}
>=	Binary	Compares for left greater than or equal to right ^{1,2}
>	Binary	Compares for left greater than right ^{1,2}
ADDR	Function	Returns the location in storage of an SBCS fixed string
HBOUND	Function	Determines the upper bound of an SBCS fixed string
LBOUND	Function	Determines the lower bound of an SBCS fixed string which is always 1
PACK	Procedure	Copies an array starting at a given point to a packed array
SIZEOF	Function	Returns the number of bytes required for an SBCS fixed string
STR	Function	Converts an SBCS fixed string to a STRING
UNPACK	Procedure	Copies a packed array to an array starting at a given point

Figure 80. Operators and Routines for the SBCS Fixed String Data Type

Notes to Figure 80:

1. If two strings being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match
2. Relative magnitude of two strings is based upon the collating sequence of EBCDIC.

SET Data Type

A variable of type SET can contain any combination of values taken from the *base scalar type*. A value is either in the set or it is not in the set. VS Pascal sets can be used in many of the same ways as bit strings (which often tend to be machine dependent). Each bit corresponds to one element of the base type and is set to a binary one when that element is a member of the set. For example, a set operation such as intersection (the operator is "&") is the same as taking the "Boolean AND" of two bit strings.

Figure 81 shows the syntax of the SET data type.

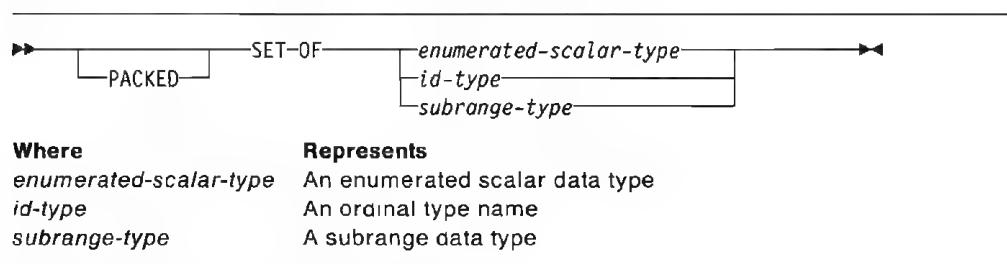


Figure 81. Syntax of the SET Data Type

Given a SET type of the form

SET OF a..b

where **a** and **b** express the lower and upper bounds of the base scalar type, the following conditions must hold:

ORD(a) >= 0
ORD(b) <= 255.

The storage and alignment required for a set variable depends on the ordinal type on which the set is based. The amount of storage required for a packed set is the minimum number of bytes needed so that every member of the set can be assigned a unique bit. Given a set definition:

TYPE
S = SET OF BASE;

where **BASE** is a scalar type that is not a subrange.

The ordinal value of the last member **M** that can be contained in the set is:

M := ORD(HIGHEST(BASE)).

The mapping of a set variable as a function of **M** would be as follows:

Range of M	Size in Bytes	Alignment
0 <= M <= 7	1	Byte
8 <= M <= 15	2	Halfword
16 <= M <= 23	3	Byte
24 <= M <= 31	4	Fullword
32 <= M <= 255	M DIV 8 + 1	Byte

A set can therefore have at most 32 bytes of storage.

The storage required for an unpacked set of a subrange is the same as that required for a set of the subrange's base type. For example, given:

TYPE
T = SET OF t;
S = SET OF s;

where **s** is a subrange of **t**. The types **T** and **S** have identical storage mappings.

Figure 82 shows an example of SET declarations.

```
TYPE
  DAYS      = (MONDAY, TUESDAY, WEDNESDAY,
               THURSDAY, FRIDAY);
  CHARS     = SET OF CHAR;
  DAYSOFMON = PACKED SET OF 1..31;
  DAYSOFWEEK = SET OF MONDAY..FRIDAY;
  FLAGS     = SET OF
               (A,B,C,D,E,F,G,H);
```

Figure 82. Example of SET Declarations

Figure 83 describes the operators and functions that apply to the SET data type. See "Predefined Routines" on page 113 for further information about these functions.

Operator or Function	Form	Description
NOT or \neg	Unary	Returns the complement of the operand
=	Binary	Compares for equality
< > or \neg =	Binary	Compares for inequality
< =	Binary	Returns TRUE if first operand is subset of second operand
> =	Binary	Returns TRUE if first operand is superset of second operand
IN	Binary	Returns TRUE if first operand (a scalar) is a member in the set represented by the second operand
+	Binary	Forms the union of two sets
*	Binary	Forms the intersection of two sets
-	Binary	Forms the difference between two sets
> < or XOR or &&	Binary	Forms the symmetric difference of two sets
ADDR	Function	Returns the location in storage of a set variable
SIZEOF	Function	Returns the number of bytes required for a value of type SET

Figure 83. Operators and Functions for Type SET

Notes to Figure 83:

- *Set complement* produces a set that has all of the elements not in the set being complemented.
- *Set union* produces a set that contains all of the elements that are members of the two operands.
- *Set intersection* produces a set that contains only the elements common to both sets.
- *Set difference* produces a set that includes all elements from the left operand except those elements in the right operand.
- *Set symmetric difference* produces a set that contains all elements from the two operands except the elements common to both operands.
- *The IN operator* tests for membership of a scalar within a set; if the scalar is not a permissible value of the set, FALSE is returned.

SHORTREAL Data Type

The predefined data type SHORTREAL represents floating-point data. Variables of this type occupy 4 bytes of storage and are aligned on a word boundary. All shortreal arithmetic is done using single-precision floating-point instructions.

Operations between data of type REAL and SHORTREAL are performed using double-precision floating-point instructions. The shortreal operand is implicitly converted to a real (see "Implicit Type Conversion" on page 46).

Because the SHORTREAL type is not ordinal, it cannot be used in certain circumstances. For example:

- In subranges or sets of SHORTREAL
- As an array index type
- As a CASE selector
- As the type of variable in a FOR loop index
- As a type of variant selector
- In the predefined functions SUCC, PRED, ORD, HIGHEST, and LOWEST.

Figure 84 describes the operators and predefined functions that apply to the SHORTREAL data type. See "Predefined Routines" on page 113 for further information about these functions.

Operator or Function	Form	Description
+	Unary	Returns the unchanged result of the operand
+	Binary	Forms the sum of the operands
-	Unary	Negates the operand
-	Binary	Forms the difference of the operands
*	Binary	Forms the product of the operands
/	Binary	Forms the floating-point quotient of the operands
=	Binary	Compares for equality
< > or \neq	Binary	Compares for inequality
<	Binary	Compares for left less than right
<=	Binary	Compares for left less than or equal to right
>=	Binary	Compares for left greater than or equal to right
>	Binary	Compares for left greater than right
ABS	Function	Returns the absolute value of a number
ADDR	Function	Returns the location in storage of a variable
ARCTAN	Function	Returns the trigonometric arctangent (in radians) of a shortreal argument
COS	Function	Returns the trigonometric cosine of a shortreal argument (in radians)
EXP	Function	Returns the value of the natural log base raised to the power of a shortreal argument
LN	Function	Returns the natural logarithm of a shortreal argument
MAX	Function	Returns the maximum value of one or more operands
MIN	Function	Returns the minimum value of one or more operands
ROUND	Function	Returns a value rounded to an integer
SIN	Function	Returns the trigonometric sine of a shortreal argument (in radians)
SIZEOF	Function	Returns the number of bytes required for a value of type SHORTREAL, which is always 4

Figure 84 (Part 1 of 2). Operators and Predefined Functions for Type SHORTREAL

Operator or Function	Form	Description
SQR	Function	Returns the square of a number
SQRT	Function	Returns the square root of a shortreal argument
TRUNC	Function	Returns a value truncated to an integer

Figure 84 (Part 2 of 2). Operators and Predefined Functions for Type SHORTREAL

SPACE Data Type

Sometimes, the need arises to represent data within storage areas that do not have the same fixed offset. One example is a directory in which each entry can be of variable-length. Another example is the processing of variable-length records from a buffer. VS Pascal provides the space structure to solve these problems. Figure 85 shows the syntax of the SPACE data type.

Where	Represents
<i>constant-expr</i>	Size of the storage area (in bytes)
<i>type</i>	Any type

Figure 85. Syntax of the SPACE Data Type

A variable of type SPACE occupies the number of bytes indicated in the length specifier of the type definition and is aligned on a byte boundary.

Figure 86 describes the functions that apply to the SPACE data type. See "Predefined Routines" on page 113 for further information about these functions.

Function	Description
ADDR	Returns the location in storage of a variable
HBOUND	Returns the declared maximum byte value of a variable
LBOUND	Returns the value 0; spaces always have a lower bound of zero
SIZEOF	Returns the number of bytes required for a value of type SPACE

Figure 86. Functions for the SPACE Data Type

STRING Data Type

The predefined data type STRING is defined as a PACKED ARRAY[1..n] OF CHAR whose length varies at run time up to a compile-time specified maximum given by a constant expression. Figure 87 shows the syntax of the STRING data type.

Where	Represents
<i>constant-expr</i>	Any integer constant expression

Figure 87. Syntax of the STRING Data Type

The value of the expression must be in the range 0..32767. The length of a string variable is initially determined when the variable is assigned a value. This length can be changed by string operators and routines. The assignment of one string to another can cause a run-time error if the actual length of the source string is greater than the maximum length of the target. VS Pascal will never automatically truncate strings.

Any variable of type STRING is compatible with any other variable of type STRING, that is, the maximum length field of a type definition has no bearing in type compatibility tests.

Note: The STRING data type manipulates both pure SBCS data and mixed SBCS/DBCS data. However, pure DBCS data is best manipulated with the GSTRING data type.

Figure 88 describes the operators and routines that apply to the STRING data type. See Figure 89 on page 83 for the functions that apply to mixed strings. The operators and routines manipulate SBCS and mixed strings in a byte-oriented manner. See "Predefined Routines" on page 113 for further information about these routines.

Operator or Routine	Form	Description
=	Binary	Compares for equality ¹
< > or \neq	Binary	Compares for inequality ¹
<	Binary	Compares for left less than right ^{1,2}
<=	Binary	Compares for left less than or equal to right ^{1,2}
>=	Binary	Compares for left greater than or equal to right ^{1,2}
>	Binary	Compares for left greater than right ^{1,2}
+ or	Binary	Concatenates the operands
ADDR	Function	Returns the location in storage of a variable
COMPRESS	Function	Returns a string with all occurrences of multiple blanks replaced by a single blank
DELETE	Function	Returns a string with a portion removed
INDEX	Function	Locates the first occurrence of a string in another string
LENGTH	Function	Returns the length of a string
LPAD	Procedure	Pads or truncates a string on the left
LTRIM	Function	Returns a string with leading blanks removed
MAXLENGTH	Function	Returns the declared length of a string
READSTR	Procedure	Reads a list of variables from a string
RINDEX	Function	Locates the last occurrence of a string in another string
RPAD	Procedure	Pads or truncates a string on the right
SIZEOF	Function	Returns the number of bytes required for a value of type STRING

Figure 88 (Part 1 of 2). Operators and Routines for Strings (Byte-Oriented)

Operator or Routine	Form	Description
STR	Function	Converts either a CHAR or an SBCS fixed string to a STRING
STOGSTR	Function	Converts a STRING to a GSTRING
SUBSTR	Function	Returns a specified portion of a string
TRIM	Function	Returns a string with trailing blanks removed
WRITESTR	Procedure	Writes a list of expressions to a string

Figure 88 (Part 2 of 2). Operators and Routines for Strings (Byte-Oriented)

Notes to Figure 88:

1. If two strings being compared are of different lengths, the shorter is assumed to be padded with blanks on the right until the lengths match.
2. Relative magnitude of two strings is based upon the collating sequence of EBCDIC.

The predefined functions listed in Figure 89 operate on mixed strings in a character-oriented manner. Character strings are validated before the operation. Adjacent shift-in/shift-out character pairs and DBCS nulls are removed from the result. This simplified string is known as a *canonical mixed string*.

Function	Description
MCOMPRESS	Returns a string with sequences of SBCS blanks replaced by one SBCS blank, and sequences of DBCS blanks replaced by one DBCS blank
MDELETE	Returns a mixed string with a specified part removed
MINDEX	Locates the first occurrence of a string in another string, manipulating SBCS and DBCS characters separately
MLENGTH	Returns the length of a mixed string
MLTRIM	Returns a string with leading SBCS and DBCS blanks removed
MRINDEX	Locates the last occurrence of a string in another string, manipulating SBCS and DBCS characters separately
MSUBSTR	Returns a specified portion of a mixed string
MTRIM	Returns a string with trailing SBCS and DBCS blanks removed

Figure 89. Functions for Strings (Character-Oriented)

Figure 90 on page 84 shows how binary operators are applied to SBCS characters, SBCS fixed strings, and SBCS strings.

Left Operand	Right Operand	Result
CHAR	CHAR	Allowed
	SBCS Fixed String	Use STR on both operands
	STRING	Use STR on the CHAR
SBCS Fixed String	CHAR	Use STR on both operands
	SBCS Fixed String	Allowed if the types are compatible
	STRING	Use STR on the fixed string
STRING	CHAR	Use STR on the CHAR
	SBCS Fixed String	Use STR on the fixed string
	STRING	Allowed

Figure 90. How to Apply Binary Operators to SBCS Characters, SBCS Fixed Strings, and SBCS Strings

Figure 91 shows how to convert strings on assignment.

Target Variable	Source Expression	Result
CHAR	CHAR	Allowed
	SBCS Fixed String	Index the fixed string to obtain a CHAR
	STRING	Index the string to obtain a CHAR
SBCS Fixed String	CHAR	Use STR
	SBCS Fixed String	Allowed if the types are compatible
	STRING	Allowed; if truncation is required, an error results
STRING	CHAR	Use STR to convert CHAR to a STRING
	SBCS Fixed String	Use STR to convert the fixed string to a STRING; if truncation is required, an error results
	STRING	Allowed; if truncation is required, an error results

Figure 91. How to Convert Strings on Assignment

STRINGPTR Data Type

The STRINGPTR data type defines the pointer to a dynamic string variable.

STRINGPTR is equivalent to:

```

TYPE
    STRINGPTR = @STRING;

```

The procedure NEW allocates storage for the STRING pointed to by the STRINGPTR. An integer expression is passed to procedure NEW in order to

specify the maximum length of the allocated string. (See "NEW Procedure" on page 143.)

Variables of type STRINGPTR have two lengths associated with them:

- The current length that defines the number of characters in the string at any instant in time
- The maximum length that defines the storage required for the string.

Figure 92 shows an example of the predefined type STRINGPTR.

```

VAR
  P      : STRINGPTR;
  Q      : STRINGPTR;
  I      : 0..32767;
BEGIN
  .
  .
  I := 30;
  NEW(P, I);
  WRITELN( MAXLENGTH(P@) );
    (*writes '30' to output*)
  NEW(Q,5);
  Q@ := '1234567890';
    (*causes a truncation*)
    (*error at execution *)
END;
```

Figure 92. Example of the Type STRINGPTR

Figure 93 describes the operators and predefined routines that apply to the STRINGPTR data type. See "Predefined Routines" on page 113 for further information about these routines.

Operator or Routine	Form	Description
=	Binary	Compares for equality
< > or \neq	Binary	Compares for inequality
ADDR	Function	Returns the location in storage of a variable
DISPOSE	Procedure	Deallocates a dynamic string variable pointed to by a pointer
DISPOSEHEAP	Procedure	Deallocates a heap
MARK	Procedure	Creates a subheap in the current heap
NEW	Procedure	Allocates a dynamic string variable with a maximum length in the current heap and sets the pointer to point to the string
NEWHEAP	Procedure	Allocates a new heap
ORD	Function	Converts an operand to an integer and returns the address of the string pointed to by a pointer

Figure 93 (Part 1 of 2). Operators and Predefined Routines for the STRINGPTR Data Type

Operator or Routine	Form	Description
QUERYHEAP	Procedure	Sets the pointer to the current heap
RELEASE	Procedure	Frees all subheaps created after a specified subheap in the heap containing the specified subheap
SIZEOF	Function	Returns the number of bytes required for a value of type STRINGPTR
USEHEAP	Procedure	Changes the current heap

Figure 93 (Part 2 of 2). Operators and Predefined Routines for the STRINGPTR Data Type

Subrange Data Type

The subrange data type is a subset of consecutive values of a previously defined ordinal. Any operation permissible on an ordinal is also permissible on any subrange of it. Figure 94 shows the syntax of the subrange data type.



Where	Represents
<i>constant</i>	Any ordinal constant
<i>constant-expr</i>	Any ordinal constant expression

Figure 94. Syntax of the Subrange Data Type

A subrange is defined by specifying the minimum and maximum values that will be permitted for data declared with that type. For subranges that are packed, VS Pascal assigns the smallest number of bytes required to represent a value of that type.

Type definitions representing integer subranges can be prefixed with the reserved word **PACKED**. VS Pascal assigns the smallest number of bytes required to represent a **PACKED** value. Given a type definition *T* as:

```
TYPE
  T = PACKED i..j;
```

the number of bytes required for different ranges of integers would be as follows. For ranges other than those listed, use the first range that encloses the desired range.

Range of <i>i..j</i>	Size	Alignment
0..255	1 byte	Byte
-128..127	1 byte	Byte
-32768..32767	2 bytes	Halfword
0..65535	2 bytes	Halfword
-8388608..8388607	3 bytes	Byte
0..16777215	3 bytes	Byte
Otherwise	4 bytes	Fullword

If the reserved word **RANGE** is used in the subrange definition, both the minimum and maximum values can be any expression that can be computed at compile time. If the reserved word **RANGE** is not used, the minimum value of the range must be a

simple constant, while the maximum value can still be any expression that can be computed at compile time.

Figure 95 shows examples of subrange scalars.

```
CONST
  SIZE      = 1000;

TYPE
  DAYS      = (SU, MO, TU, WE,
               TH, FR, SA);
  MONTHS    = (JAN, FEB, MAR, APR,
               MAY, JUN, JUL, AUG,
               SEP, OCT, NOV, DEC);
  UPPERCASE  = 'A' .. 'Z';
  HUNDRED    = 0 .. 99;
  CODES      = RANGE
               CHR(0)..CHR(255);
  INDEX      = PACKED 1 .. SIZE+1;

VAR
  WORKDAY    : MO .. FR;
  SUMMER     : JUN .. AUG;
  SMALLINT   : PACKED 0..255;
  YEAR       : 1900 .. 2000;
```

Figure 95. Examples of Subrange Scalars

Figure 96 illustrates that two subrange types can be defined over the same base type. Operations are permitted between these two variables because they have the same base type.

```
VAR
  NEG        : MININT .. -1;
  POS        : 1 .. MAXINT;
```

Figure 96. Examples of Subranges with the Same Base Type

Restrictions:

- A subrange of type **GCHAR**, **REAL**, or **SHORTREAL** is not permitted.
- The number of values in a subrange of type **CHAR** is determined by the collating sequence of the EBCDIC character set.

Figure 97 on page 88 describes the predefined functions that apply to the subrange scalar data type. See "Predefined Routines" on page 113 for further information about these procedures and functions.

Function	Description
HIGHEST	Returns the maximum value of a subrange type
LOWEST	Returns the minimum value of a subrange type
MAX	Returns the maximum value of one or more scalar expressions
MIN	Returns the minimum value of one or more scalar expressions
ORD	Converts an ordinal expression to an integer
PRED	Returns the predecessor of an ordinal expression
SUCC	Returns the successor of an ordinal expression

Figure 97. Predefined Functions for Type Subrange Scalar

TEXT Data Type

In Standard Pascal, the predefined data type TEXT is defined as:

```
TYPE
  TEXT = FILE OF CHAR;
```

VS Pascal predefines the two TEXT variables INPUT and OUTPUT. Because they are predefined, they do not need to be explicitly declared in your program. You do not need to include the files INPUT and OUTPUT in the program header even if they are used in the program.

In addition to the routines that allow character input and output operations, there are several routines that perform line-oriented input and output, as well as conversions between character and internal (binary) representations.

Figure 98 describes the predefined routines that apply to the TEXT data type. See "Predefined Routines" on page 113 for further information about these routines.

Routine	Form	Description
ADDR	Function	Returns the location in storage of a variable
CLOSE	Procedure	Closes a file
COLS	Function	Returns the current column of a file
EOF	Function	Tests a file for end-of-file condition
EOLN	Function	Tests a file for end-of-line condition
GET	Procedure	Reads the current character of a file and advances the file pointer to the next character in the input file
PAGE	Procedure	Skips to the top of the next page
PDSIN	Procedure	Opens a file for input, specifying open options and the PDS member name
PDSOUT	Procedure	Opens a file for output, specifying open options and the PDS member name
PUT	Procedure	Writes the file buffer to a file and advances the file pointer to the next character in the output file

Figure 98 (Part 1 of 2). Procedures and Functions for Type TEXT

Routine	Form	Description
READ	Procedure	Reads from a file into a variable
READLN	Procedure	Reads a variable and then skips to end-of-line of a text file
RESET	Procedure	Opens a file for input with the open options
REWRITE	Procedure	Opens a file for output with the open options
SIZEOF	Function	Returns the number of bytes required for a value of type TEXT
TERMIN	Procedure	Opens a file for input from the terminal with the open options
TERMOUT	Procedure	Opens a file for output for the terminal with the open options
WRITE	Procedure	Writes a value to a file
WRITELN	Procedure	Writes a value and then writes an end-of-line to a text file

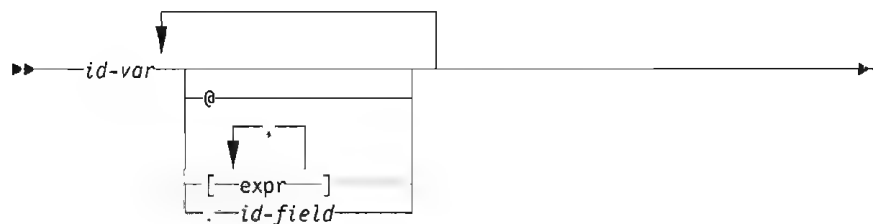
Figure 98 (Part 2 of 2). Procedures and Functions for Type TEXT

Chapter 7. Variables

VS Pascal divides variables into five groups, according to how they are declared:

- Automatic (VAR variables)
- Dynamic (pointer-qualified variables)
- **Static (static variables)**
- **External (DEF/REF variables)**
- Parameter (declared in a routine header).

A variable can be referenced in several ways depending on its type. You can always refer to the entire variable by specifying its name. You can refer to a dynamic variable or a component of a structured variable by using the syntax shown in Figure 99. Assignments of entire records or arrays are allowed.



Where	Represents
<i>id-var</i>	Name of variable
<i>@</i>	Pointer or file reference
<i>expr</i>	Subscripted variable reference (index expression)
<i>id-field</i>	Field reference

Figure 99. Syntax of a Variable Reference

Figure 100 shows an example of variables used in their entirety.

```
VAR
  LINE1,
  LINE2 : PACKED
          ARRAY[ 1..80 ] OF
            CHAR;
.
.
BEGIN
  (*assign all 80 characters *)
  (*of the array             *)
  LINE1 := LINE2;
END;
```

Figure 100. Example of Variables Used in Their Entirety

Predefined Variables

VS Pascal has two predefined variables:

- INPUT, the default input file
- OUTPUT, the default output file.

Subscripted Variables

Array Variables

An element of an array is selected by placing an indexing expression, enclosed within square brackets, after the name of the array. The indexing expression must be assignment compatible with the data type declared in the array's index definition. See "Compatible Data Types" on page 47 for further information on compatibility.

A multidimensional array can be referenced as an array of arrays. For example, let variable A be declared as follows:

```
A: ARRAY [a..b,c..d] OF T
```

As explained in "ARRAY Data Type" on page 51, this declaration is equivalent to:

```
A: ARRAY [a..b] OF  
    ARRAY [c..d] OF T
```

A reference of the form A[I] is a variable of type:

```
ARRAY [c..d] OF T
```

and represents a single row in array A. A reference of the form A[I][J] is a variable of type T and represents the Jth element of the Ith row of array A. This latter reference is customarily abbreviated as:

```
A[I,J]
```

Any array reference with two or more subscript indexes can be abbreviated by writing the subscripts in order, separated by commas. That is, A[I][J]... can be written as A[I,J,...].

Figure 101 on page 94 shows an example of array indexing.

```

TYPE

    MATRIX = ARRAY[1..10, 1..10] OF REAL;

    MATRISO = ARRAY[1..10] OF          (* An alternative declaration *)
                ARRAY[1..10] OF REAL;  (* for MATRIX, above.      *)

    COLOR   = (RED, YELLOW, BLUE);

    INTENSITY = PACKED ARRAY [COLOR] OF REAL;

VAR
    M          : MATRIX;
    HUE         : INTENSITY;

BEGIN

    (*assign ten element array*)
    M[1]        := M[2];

    (*assign one element of a two*)
    (*dimensional array two ways *)
    M[1,1]      := 3.14159;
    M[1][1]     := 3.14159;

    (*this is a reddish orange *)
    HUE[RED]    := 0.7;
    HUE[YELLOW] := 0.7;
    HUE[BLUE]   := 0.7;
END;

```

Figure 101. Example of Array Indexing

STRING Variables

A variable of type **STRING** can be subscripted with an integer expression to reference individual characters. The value of the subscript must not be less than 1 or greater than the length of the string. Subscripting a **STRING** returns a **CHAR**.

Because strings are not true arrays, you must use the long form of subscripting with arrays of type **STRING**. Figure 102 shows examples of valid and invalid subscripting for a **STRING** data type.

```

VAR
    A : ARRAY [1..10] OF STRING;
    C : CHAR;
BEGIN
    .
    .
    C := A[5,1];  (* Not legal *)
    C := A[5][1]; (* Legal    *)
    .
    .
END;

```

Figure 102. Examples of Valid and Invalid Subscripting for a **STRING** Data Type

GSTRING Variables

A variable of type GSTRING can be subscripted with an integer expression to reference individual characters. The value of the subscript must not be less than 1 or greater than the length of the string. Subscripting a GSTRING returns a GCHAR.

A variable of type GSTRING is indexed in a character-oriented manner; that is, an index is expressed in terms of DBCS characters. Because GSTRINGs, like strings, are not true arrays, you must use the long form of subscripting with arrays of GSTRINGs. Figure 103 shows an example of GSTRING indexing.

```
VAR
  A : ARRAY [1..10] OF GSTRING;
  C : GCHAR;
BEGIN
  .
  .
  C := A[5,1];  (* Not legal *)
  C := A[5][1]; (* Legal *)
  .
  .
END;
```

Figure 103. Example of GSTRING Indexing

Error Checking

When the %CHECK SUBSCRIPT option is enabled, VS Pascal checks the index expression at run time to ensure its value lies within the subscript range of the array or string. VS Pascal issues a run-time error message if the value lies outside of the prescribed range. (For a description of the CHECK feature, see “%CHECK Directive” on page 231.)

Field Referencing

To select a field of a record, write:

- The record variable name
- A period
- The name of the field.

Figure 104 on page 96 shows an example of field referencing.

```

VAR
  PERSON:
    RECORD
      FIRSTNAME,
      LASTNAME: STRING(15);
    END;

  DATE:
    RECORD
      DAY: 1..31;
      MONTH: 1..12;
      YEAR: 1900..2000
    END;

  I: INTEGER;
  DECK:
    ARRAY[1..52] OF
      RECORD
        CARD: 1..13;
        SUIT:
          (SPADE, HEART,
           DIAMOND, CLUB)
      END;
  BEGIN
    I := 1;
    PERSON.FIRSTNAME := 'STEVE';
    PERSON.LASTNAME := 'MUELLER';
    DATE.YEAR := 1978;
    DECK[ I ].CARD := 2;
    DECK[ I ].SUIT := SPADE;
  END;

```

Figure 104. Example of Field Referencing

Pointer Referencing

A dynamic variable is created by the predefined procedure **NEW** or by an implementation-provided routine that assigns an address to a pointer variable. You can refer either to the pointer or to the dynamic variable; referencing the dynamic variable requires pointer notation, also called “dereferencing” the pointer.

For example, given the declaration:

```
VAR P : @ R;
```

P refers to the pointer

P@ refers to the dynamic variable

a reference to P is a reference to the pointer, and a reference to P@ is a reference to the dynamic variable.

Figure 105 on page 97 shows an example of pointer referencing.

```
TYPE
  INFO = RECORD
    AGE: 1..99;
    WEIGHT: 1..400;
  END;

  FAMILY =
  RECORD
    FATHER,
    MOTHER,
    SELF: @INFO;
    KIDS: 0..20
  END;

VAR
  FAMILYPOINTER : @FAMILY;
BEGIN
  NEW(FAMILYPOINTER);
  FAMILYPOINTER.KIDS := 2;
  NEW(FAMILYPOINTER.FATHER);
  FAMILYPOINTER.FATHER.AGE := 35;
END;
```

Figure 105. Example of Pointer Referencing

When the %CHECK POINTER option is enabled, VS Pascal issues a run-time error message when an attempt is made to reference a pointer that has the value NIL. (For a description of the CHECK feature, see "%CHECK Directive" on page 231.)

File Referencing

A component of a file is selected from the file buffer by a pointer notation. The file variable is assigned by using the predefined procedures GET and PUT. Each call of these procedures moves the current component to the output file (PUT) or assigns a new component from the input file (GET). For a description of GET and PUT, see "GET Procedure" on page 126 and "PUT Procedure" on page 153.

Figure 106 on page 98 shows an example of file referencing.

```

VAR
  INPUT    : TEXT;
  OUTPUT   : TEXT;
  LINE1    : ARRAY [1..80] OF CHAR;
  I        : INTEGER;
  .
  .
  (*scan off blanks          *)
  (*from a file of CHAR      *)
  GET(INPUT);
  WHILE INPUT@ = ' ' DO
    GET(INPUT);

  (*transfer a line to the    *)
  (*OUTPUT file               *)
  FOR I := 1 TO 80 DO
    BEGIN
      OUTPUT@ := LINE1[I];
      PUT(OUTPUT);
    END;

```

Figure 106. Example of File Referencing

Space Referencing

A variable declared with the `SPACE` data type has a component that can “float” over a storage area in a byte-oriented manner. A component of a space is selected by placing an index expression, enclosed within square brackets, after the space variable (just as in array references). The indexing expression must be an `INTEGER` type or a subrange of `INTEGER`. The value of the index is the offset within the space where the component is to be accessed. The unit of the index is the byte. The index is always based upon a zero origin; the index range of the space is from zero to one less than the value of the constant expression in the space declaration. The component is the space base type.

VS Pascal allows you to pass an element of a space by `VAR`, but it does issue a warning message when you do so.

Figure 107 on page 99 shows examples of space referencing.

```

VAR
  (*declare a space variable
   with index range 0..99 *)
  S: SPACE[100] OF
    RECORD
      A,B: INTEGER;
    END;

BEGIN
  (*base record begins
   at offset 10 within
   space *)
  S[10].A := 26;
  S[10].B := 0;
END;

```

Figure 107 Examples of Space Referencing

When the %CHECK SUBSCRIPT option is enabled, VS Pascal checks the index expression at run time to ensure that the computed address lies within the storage occupied by the space. VS Pascal issues a run-time error message if the value is invalid. (For a description of the CHECK feature, see "%CHECK Directive" on page 231).

No check is made for values extending past the end of a space; VS Pascal can act unpredictably when such an assignment or reference is attempted.

Figure 108 shows an example of invalid space referencing.

```

VAR
  S: SPACE[100] OF INTEGER;
  I: INTEGER;
BEGIN
  S[98] := I;      (* Invalid-extends past end of space *)
END;

```

Figure 108. Example of Invalid Space Referencing

Chapter 8. Routines

Procedures and functions, known collectively as routines, are the building blocks of a VS Pascal program. Procedures and functions define blocks of statements to be executed as a unit each time the procedure or function is invoked.

Procedures can be thought of as adding new statements to VS Pascal. These statements help you program more quickly because they are tailored to your needs.

Functions can be thought of as adding new operators to VS Pascal. These operators help you program more quickly by manipulating data exactly as you want.

Procedures and functions return data:

- Through the function results
Note: A function should return data *only* through its result, and not through VAR parameters or global assignments.
- Through VAR parameters
- By assigning values to variables outside the lexical scope of the routine making the assignment.

Note: These variables can also be said to be global to the routine.

You can nest routines up to eight levels deep (including the main program).

The following sections describe routine declarations, parameters and directives, and every VS Pascal routine in alphabetical order.

Routine Declarations

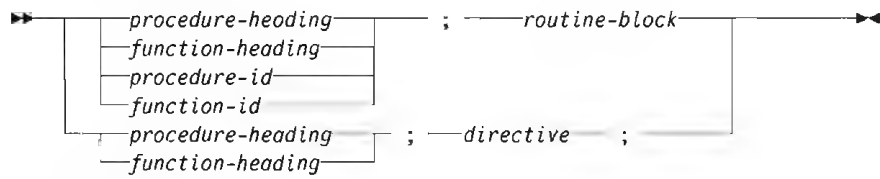
You must declare routines before they are used. A routine declaration consists of:

- A routine heading
- Declarations of local labels and identifiers
- A compound statement.

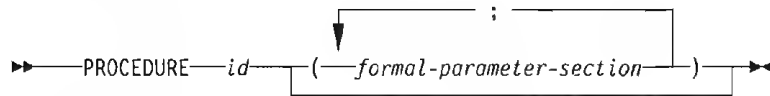
The heading defines the name of the routine and binds the formal parameters to the routine. The heading of a function declaration also binds the function name to the type of value returned by the function. Formal parameters specify data to be passed to the routine when it is invoked. The declarations are described in Chapter 4, "Declarations" on page 24. The compound statement will be executed when the routine is invoked.

Figure 109 on page 103 shows the syntax of routine declarations.

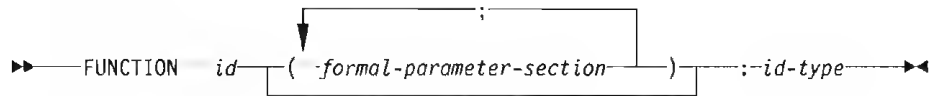
Routine Declaration



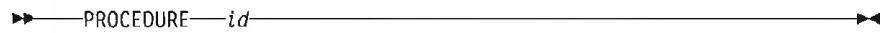
Procedure Heading



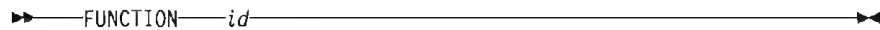
Function Heading



Procedure-id



Function-id



Routine Block

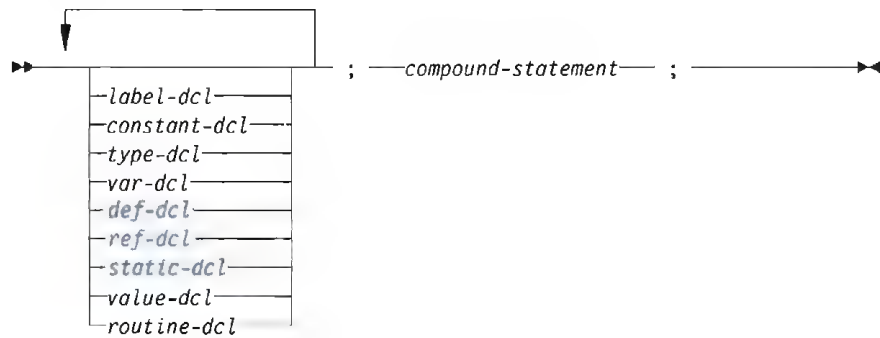


Figure 109 (Part 1 of 2). Syntax of Routine Declarations

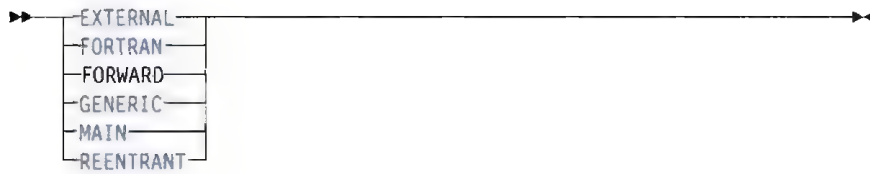
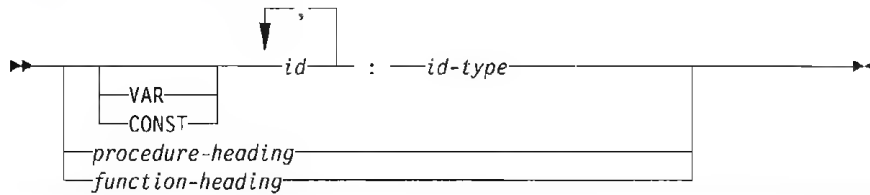
Directive**Formal Parameter Section**

Figure 109 (Part 2 of 2). Syntax of Routine Declarations

Examples of routine declarations are shown in Figure 110.

```
STATIC
  C: CHAR;

FUNCTION GETCHAR:CHAR;
  EXTERNAL;

PROCEDURE EXPR(VAR VAL: INTEGER);
  EXTERNAL;

PROCEDURE FACTOR(VAR VAL: INTEGER);
  EXTERNAL;

PROCEDURE FACTOR;
BEGIN
  C := GETCHAR;
  IF C = '(' THEN
    BEGIN
      C := GETCHAR;
      EXPR(VAL)
    END
  ELSE
    .
  .
END;

PROCEDURE EXPR (*VAR VAL: INTEGER*);
BEGIN
  FACTOR(VAL);
  .
  .
END;
```

Figure 110. Examples of Routine Declarations

Routine Parameters

When a routine is defined, *formal parameters* are bound to it. Formal parameters define the way and the type of data that will be passed to a routine when it is invoked. A procedure or function can be passed to a routine as a formal parameter. Within the called routine, the formal parameter can be used as if it were a procedure or function.

When the routine is invoked, a parameter list is built. At the point of invocation, the parameters are called the *actual parameters*. Actual and formal parameters cannot be declared as anonymous types.

VS Pascal permits parameters to be passed in the following ways:

- Pass-by-value
- Pass-by-read/write-reference (VAR)
- Pass-by-read-only-reference (CONST)
- Formal routine parameter.

Pass-by-Value Parameters

Pass-by-value parameters can be thought of as local variables initialized by the caller. The called routine can change the value of this kind of parameter, but the change is never reflected back to the caller. Any expression, variable, or constant can be passed with this mechanism. The exception is that the parameter cannot be of type FILE, or of any type that even indirectly contains a FILE type. Actual parameters must be assignment compatible with the formal parameters.

Pass-by-VAR Parameters

Pass-by-VAR (variable) parameters are also called *pass-by-read/write reference* parameters. Parameters passed by VAR reflect modifications to the parameters back to the caller. Therefore, you can use this parameter type as both an input and output parameter. The use of the VAR symbol in a parameter indicates that the parameter is to be passed by read/write reference. Only variables can be passed by this mechanism; expressions and constants cannot be passed in this way.

Fields of a packed record or elements of a fixed string can be passed as VAR parameters in VS Pascal (even though this is not allowed in Standard Pascal).

Actual scalar and set VAR parameters need only have compatible types of the same size in VS Pascal (in Standard Pascal they must have the same type). These parameters will be checked for assignment compatibility.

It is often desirable to call a procedure or function and pass in a string whose declared length does not match that of the formal parameter. The *conformant string parameter* is used for this purpose.

The conformant string parameter is a pass-by-VAR parameter with a type specified as STRING without a length qualifier. Strings of any declared length will conform to such a parameter. You can use the MAXLENGTH function to obtain the declared length (see "MAXLENGTH Function" on page 137). See Figure 111 on page 106 for an example of conformant string parameters.

Pass-by-CONST Parameters

Pass-by-CONST parameters cannot be altered by the called routine. Also, you must not modify the actual parameter value until the call is completed. If you attempt to alter the actual parameter while it is being passed by CONST, VS Pascal can act unpredictably. This method is called *pass-by-read-only-reference*. The parameters appear to be constants from the called routine's point of view. Any expression, variable, or constant can be passed by CONST (fields of a packed record and elements of a packed array can also be passed). The use of the CONST reserved word in a parameter indicates that the parameter is to be passed by this mechanism. With parameters that are structures (such as strings), passing by CONST is usually more efficient than passing by value. Actual parameters must be assignment compatible with the formal parameters.

It is often desirable to call a procedure or function and pass in a string whose declared length does not match that of the formal parameter. The *conformant string parameter* is used for this purpose.

The conformant string parameter is a pass-by-CONST parameter with a type specified as STRING without a length qualifier. Strings of any declared length will conform to such a parameter. Figure 111 shows an example of conformant string parameters.

```
PROCEDURE TRANSLATE
  (VAR S      : STRING;
   CONST TABLE : STRING;
  VAR
    I  : 0..32767;
    J  : 1..ORD(MAXCHAR) + 1;
  BEGIN
    FOR I := 1 TO LENGTH(S) DO
      BEGIN
        J := ORD(S[I]) + 1;
        IF J > LENGTH(TABLE) THEN
          S[I] := ' ';
        ELSE
          S[I] := TABLE[J];
        END;
      END;
    END;
```

Figure 111. Example of Conformant String Parameters

Formal Routine Parameters

A procedure or function can be passed to a routine as a formal parameter. Within the called routine, the formal parameter can be used as if it were a procedure or function.

When using actual and formal routine parameters, both routines must be either procedures, or functions with the same result type. In addition, the formal parameter lists of the actual and formal routine parameters must be *congruous*. Parameter lists are congruous when:

- Both lists contain the same number of formal parameter sections
- Formal parameter sections in corresponding positions have the same number of matching formal parameters.

In VS Pascal, formal parameter sections are ignored. Parameter lists need only have the same number of formal parameters, and formal parameters in corresponding positions must match. In VS Pascal, formal parameters match when both are:

- Value parameters with the same type
- VAR parameters with the same type
- **CONST** parameters with the same type
- Procedural parameters with congruous parameter lists
- Functional parameters with congruous parameter lists and the same result type.

Routines That Can Be Passed as Parameters

Standard Pascal does not allow any predefined routines to be passed as parameters.

VS Pascal allows some predefined routines to be passed as parameters to another routine. These routines are listed in Figure 112.

ARCTAN	COS	HALT	PARMS	SIN	TRACE
CLOCK	DATETIME	LN	RANDOM	SQRT	
COLS	EXP	LTOKEN	RETCODE	TOKEN	

Figure 112. Predefined Routines That Can Be Passed as Parameters

VS Pascal allows other routines (which are not predefined) to be passed as parameters to another routine. These routines are listed in Figure 113.

CMS	ONERROR
ITOHS	PICTURE

Figure 113. Routines That Can Be Passed as Parameters

Note to Figure 113: ITOHS and PICTURE can be passed as parameters only when the %INCLUDE CONVERT form is *not* used, and a type name for STRING(x) is defined.

Restriction: GENERIC procedures and FORTRAN functions or subroutines cannot be passed as parameters to a VS Pascal routine.

Function Results

A value is returned from a function by assigning the value to the name of the function before leaving the function. This value is inserted within the expression at the point of the call. The value must be assignment compatible with the declared function type.

If the function name is used on the right side of an assignment, it will be interpreted as a recursive call. Figure 114 on page 108 shows an example of a recursive function.

```

FUNCTION FACTORIAL (X: INTEGER): INTEGER;
BEGIN
  IF X <= 1 THEN
    FACTORIAL := 1
  ELSE
    FACTORIAL := X * FACTORIAL(X-1)
  END;

```

Figure 114. Example of a Recursive Function

Standard Pascal permits a function to return only a scalar or pointer value. In VS Pascal, a function can return any type except a file or any type containing a file. This means that you can write a VS Pascal function that returns a record structure as its result (you might wish to do this for implementing a complex arithmetic library). A function can also return a string. However, you must specify the maximum length of the string to be returned.

Figure 115 shows an example of a function returning a record.

```

TYPE
  COMPLEX = RECORD
    R,I : REAL
  END;

FUNCTION CADD (CONST A,B : COMPLEX) : COMPLEX;
VAR
  C : COMPLEX;
BEGIN
  C.R := A.R + B.R;
  C.I := A.I + B.I;
  CADD := C;
END;

```

Figure 115. Example of a Function Returning a Record

Routine Directives

Routine directives allow you to declare routines that have special properties. There are three categories of routine directive:

- | | |
|---------------------|--|
| Identified | <p>There must be a routine identification specifying the routine's block within the immediately enclosing routine.</p> <p>The FORWARD, MAIN, and REENTRANT routine directives are identified.</p> |
| Unidentified | <p>There must <i>not</i> be a routine identification specifying the routine's block within the immediately enclosing routine.</p> <p>The FORTRAN and GENERIC routine directives are unidentified.</p> |
| Contextual | <p>The routine assumes different properties depending on whether the routine identification specifying the routine's block is within the immediately enclosing routine.</p> <p>The EXTERNAL routine directive is contextual.</p> |

EXTERNAL Routine Directive

EXTERNAL identifies a procedure or function that can be invoked from outside of its lexical scope (such as another unit). The EXTERNAL routine directive is used to specify the heading of such a routine directive. While many units can call an EXTERNAL routine, only one unit will actually contain the body of the routine. The formal parameters defined in the EXTERNAL routine declaration must match those in the unit where the routine is defined. An EXTERNAL routine declaration can refer to a VS Pascal routine located later in the same unit, or located in another unit, or it can refer to code produced by other means (such as assembler code).

If an EXTERNAL routine is identified, the body of the routine must be declared in the outermost nesting level of a unit; that is, it must not be nested in another routine. Such a routine is an entry point in the unit.

Figure 116 illustrates two units (a program unit and a segment unit) that share a single EXTERNAL routine. Both units can invoke the routine, but only one unit contains the definition of the routine.

```
PROGRAM TEST;
  FUNCTION SQUARE(X : REAL) : REAL;
    EXTERNAL;
BEGIN
  WRITELN( SQUARE(44) );
END .

SEGMENT S;
FUNCTION SQUARE(X : REAL) : REAL;
  EXTERNAL;
FUNCTION SQUARE;
BEGIN
  SQUARE := X * X;
END; .
```

Figure 116. Example of the EXTERNAL Directive

See *VS Pascal Application Programming Guide* for more information on the EXTERNAL routine directive.

FORTRAN Routine Directive

FORTRAN, like EXTERNAL, identifies a routine defined outside the unit being compiled. In addition, it specifies that the routine is not written in Pascal. Therefore, you must use the conventions of FORTRAN. In order to meet the requirements of FORTRAN, you must obey the restrictions listed below.

Restrictions:

- All parameters can be only VAR or CONST parameters. If you pass a parameter by CONST to a FORTRAN routine, you must ensure that the FORTRAN routine does not alter the contents of the parameter.
- If the FORTRAN routine is a function, it can return only a scalar result (this includes real and shortreal).
- Routines cannot be passed to a FORTRAN routine.
- Multidimensional arrays are not remapped to conform to FORTRAN indexing. An element of an array $A[n,m]$ in Pascal will be element $A(m,n)$ in FORTRAN.

Restrictions:

- Only procedures can be declared as GENERIC
- Pass-by-value parameters cannot
- A GENERIC routine cannot
(GENERIC routine cannot
of routine)

Predefined Routines

Data Inquiry Routines

Routine Name	Routine Type	Description	See Page
ADDR	Function	Returns the address of a variable	118
HBOUND	Function	Returns the upper bound of a fixed string	128
HIGHEST	Function	Returns the maximum value of an ordinal type	129
LBOUND	Function	Returns the lower bound of a fixed string	130
LOWEST	Function	Returns the minimum value of an ordinal type	132
MAX	Function	Returns the maximum value of a list of scalars	137
MIN	Function	Returns the minimum value of a list of scalars	139
ODD	Function	Returns TRUE if the integer argument is odd	149
PRED	Function	Obtains the predecessor of an ordinal type	152
SIZEOF	Function	Returns the storage size of a variable or type	167
SUCC	Function	Obtains the successor of an ordinal type	171

Figure 120. Summary of Data Inquiry Routines

Data Movement Routines

Routine Name	Routine Type	Description
PACK	Procedure	Copies an array to a packed array
UNPACK	Procedure	Copies a packed array to an array

Figure 121. Summary of Data Movement Routines

General Routines

Routine Name	Routine Type	Description
TRACE	Procedure	Writes the routine return stack
HALT	Procedure	Stops program execution

Figure 122. Summary of General Routines

Input/Output Routines

Routine Name	Routine Type	Record or TEXT	Description
	Procedure	Both	Closes a file

EXTERNAL Routine Directive

EXTERNAL identifies a procedure or function that can be invoked from outside of its lexical scope (such as another unit). The EXTERNAL routine directive is used to specify the heading of such a routine directive. While many units can call an EXTERNAL routine, only one unit will actually contain the body of the routine. The formal parameters defined in the EXTERNAL routine declaration must match those in the unit where the routine is defined. An EXTERNAL routine declaration can refer to a VS Pascal routine located later in the same unit, or located in another unit, or it can refer to code produced by other means (such as assembler code).

If an EXTERNAL routine is identified, the body of the routine must be declared in the outermost nesting level of a unit; that is, it must not be nested in another routine. Such a routine is an entry point in the unit.

Figure 116 illustrates two units (a program unit and a segment unit) that share a single EXTERNAL routine. Both units can invoke the routine, but only one unit contains the definition of the routine.

```
PROGRAM TEST;
  FUNCTION SQUARE(X : REAL) : REAL;
    EXTERNAL;
  BEGIN
    WRITELN( SQUARE(44) );
  END .

SEGMENT S;
  FUNCTION SQUARE(X : REAL) : REAL;
    EXTERNAL;
  FUNCTION SQUARE;
  BEGIN
    SQUARE := X * X;
  END; .
```

Figure 116. Example of the EXTERNAL Directive

See *VS Pascal Application Programming Guide* for more information on the EXTERNAL routine directive.

FORTRAN Routine Directive

FORTRAN, like EXTERNAL, identifies a routine defined outside the unit being compiled. In addition, it specifies that the routine is not written in Pascal. Therefore, you must use the conventions of FORTRAN. In order to meet the requirements of FORTRAN, you must obey the restrictions listed below.

Restrictions:

- All parameters can be only VAR or CONST parameters. If you pass a parameter by CONST to a FORTRAN routine, you must ensure that the FORTRAN routine does not alter the contents of the parameter.
- If the FORTRAN routine is a function, it can return only a scalar result (this includes real and shortreal).
- Routines cannot be passed to a FORTRAN routine.
- Multidimensional arrays are not remapped to conform to FORTRAN indexing. An element of an array $A[n,m]$ in Pascal will be element $A(m,n)$ in FORTRAN.

- The body of a FORTRAN routine cannot be written in Pascal.

See *VS Pascal Application Programming Guide* for more information on the FORTRAN routine directive.

FORWARD Routine Directive

FORWARD identifies a routine whose head is being declared in advance of its body. The declaration consists only of the routine heading, followed by the FORWARD routine directive. To declare the routine's body, once again declare the routine heading, but omit the formal parameter list. If the routine being declared is a function, you must also omit the function result type.

Declaring a routine FORWARD lets you call that routine before actually defining it. This is particularly useful when two routines are mutually recursive and reside at the same nesting level; one of the two must be declared FORWARD.

GENERIC Routine Directive

GENERIC identifies routines from other software products, such as IMS, that can be called by VS Pascal. This allows calls to routines that allow multiple parameter list formats. This includes routines that require different data types depending on the function being performed and routines that allow varying numbers of parameters.

Like FORTRAN and EXTERNAL, GENERIC identifies a routine defined outside the unit being compiled. The routine declaration cannot contain formal parameters. A GENERIC routine's parameters are "declared" only when the routine is called. Therefore,

```
PROCEDURE P; GENERIC;
```

is a legal declaration, but

```
PROCEDURE Q(VAR I : INTEGER); GENERIC;
```

is not because the declaration contains a formal parameter list.

To pass an actual parameter to a GENERIC procedure, specify:

1. The parameter-passing mechanism in the form of a formal parameter list
2. The actual parameter to be passed.

Figure 117 on page 111 shows how a GENERIC routine is declared and then called within the body of a program. Note that the call to the GENERIC procedure P contains the parameter-passing mechanism, followed by the actual parameter to be passed.

```

PROGRAM POLYMORPHIC;
TYPE
  FUNCTIONS = (DEVICE_QUERY, PAINT_SCREEN, DRAW_LINE, DRAW_POLY);
  COLORS = (RED, GREEN, BLUE, YELLOW, ORANGE, PURPLE);

VAR
  ROWS, COLS : INTEGER;

PROCEDURE DRAW; GENERIC;

BEGIN
  DRAW(CONST DEVICE_QUERY, VAR ROWS, VAR COLS);
  .
  .
  DRAW(CONST PAINT_SCREEN, CONST BLUE);
END.

```

Figure 117. Example of the GENERIC Routine Directive

Figure 118 shows an example of the coding required before development of the **GENERIC** routine directive. The same function is obtained by declaring the procedure being called with different parameter lists in two different routines.

```

PROGRAM POLYMORPHIC;
TYPE
  FUNCTIONS = (DEVICE_QUERY, PAINT_SCREEN, DRAW_LINE, DRAW_POLY);
  COLORS = (RED, GREEN, BLUE, YELLOW, ORANGE, PURPLE);

VAR
  ROWS, COLS : INTEGER;

PROCEDURE DEV_QUERY(VAR ROWS, COLS : INTEGER);
  PROCEDURE DRAW(CONST FUNC : FUNCTIONS;
                 VAR ROWS, COLS : INTEGER);
    FORTRAN;
  BEGIN
    DRAW(DEVICE_QUERY, ROWS, COLS);
  END;

PROCEDURE PAINT_SCR(CONST COLOR : COLORS);
  PROCEDURE DRAW(CONST FUNC : FUNCTIONS;
                 CONST COLOR : COLORS);
    FORTRAN;
  BEGIN
    DRAW(PAINT_SCREEN, COLOR);
  END;

BEGIN
  DEV_QUERY(ROWS, COLS);
  .
  .
  PAINT_SCR(BLUE);
END.

```

Figure 118. Example of Coding Before Development of the GENERIC Routine Directive

Restrictions:

- Only procedures can be declared as **GENERIC**.
- Pass-by-value parameters cannot be passed to **GENERIC** routines.
- A **GENERIC** routine cannot be passed as a parameter to another routine. (GENERIC routines do not have fixed-format parameter lists, which is required of routines passed to other routines.)
- The body of a **GENERIC** routine cannot be written in Pascal.

See *VS Pascal Application Programming Guide* for more information on the **GENERIC** routine directive.

MAIN Routine Directive

MAIN identifies a Pascal procedure that can be invoked as if it were a main program. It is sometimes desirable to invoke a VS Pascal procedure from a non-Pascal routine, such as FORTRAN or assembler language. In this case, it is necessary for certain initializing operations to be performed before actually executing the Pascal procedure. The **MAIN** directive specifies that the appropriate actions are to be performed.

Restrictions:

- The execution of a **MAIN** procedure *cannot* be reentrant.
- Only procedures can have the **MAIN** directive.
- A **MAIN** procedure's declaration and body must be in the same unit.
- The **MAIN** directive can be applied only to procedures in the outermost nesting level of a unit.
- Because the **MAIN** directive simulates a program invocation, a **MAIN** procedure must not reference a unit's global variables. Unpredictable results can occur if a **MAIN** procedure references a unit's global variables.

See *VS Pascal Application Programming Guide* for more information on the **MAIN** routine directive.

REENTRANT Routine Directive

REENTRANT identifies a Pascal procedure that can be invoked as if it were a main program, like a **MAIN** procedure. In addition, invocations of these procedures will be reentrant.

In order to achieve a reentrant invocation the first parameter of a procedure defined with the **REENTRANT** directive must be an **INTEGER** passed by **VAR**. Before the first call from a non-VS Pascal program, you must initialize this variable to zero. On subsequent calls, you must pass the same variable back unaltered (VS Pascal sets the variable on the first call and needs that value on the subsequent invocations). You need not call the same procedure each time; you can call different procedures as long as you continue to pass this variable on each call.

Restrictions:

- Only procedures can have the REENTRANT directive.
- A REENTRANT procedure's declaration and body must be in the same unit.
- The REENTRANT directive can be applied only to procedures in the outermost nesting level of a unit.
- Because the REENTRANT directive simulates a program invocation, a REENTRANT procedure must not reference a unit's global variables. Unpredictable results can occur if a REENTRANT procedure references a unit's global variables.

Note: All VS Pascal internal procedures and functions are reentrant. The REENTRANT directive identifies a procedure that is reentrant and that can be invoked from outside the VS Pascal run-time environment.

See *VS Pascal Application Programming Guide* for more information on the REENTRANT routine directive.

Predefined Routines

VS Pascal provides a wide range of predefined functions and procedures. The following sections describe these routines in detail.

Figure 119 through Figure 128 present all predefined routines by function, with a brief explanation of each. The descriptions of all functions follow in alphabetic order, starting on page 118.

Conversion Routines

Routine Name	Routine Type	Description	See Page
CHR	Function	Converts an integer to a character value	119
FLOAT	Function	Converts an integer to a floating-point value	126
GSTR	Function	Converts a GCHAR or DBCS fixed string to a GSTRING	126
GTOSTR	Function	Converts a GSTRING to a STRING	127
ORD	Function	Converts an ordinal expression or pointer to an integer	150
ROUND	Function	Converts a floating-point number to an integer by rounding	165
STOGSTR	Function	Converts a STRING to a GSTRING	168
STR	Function	Converts a CHAR or fixed string to a STRING	169
TRUNC	Function	Converts a floating-point number to an integer by truncating	174

Figure 119. Summary of Conversion Routines

Data Inquiry Routines

Routine Name	Routine Type	Description	See Page
ADDR	Function	Returns the address of a variable	118
HBOUND	Function	Returns the upper bound of a fixed string	128
HIGHEST	Function	Returns the maximum value of an ordinal type	129
LBOUND	Function	Returns the lower bound of a fixed string	130
LOWEST	Function	Returns the minimum value of an ordinal type	132
MAX	Function	Returns the maximum value of a list of scalars	137
MIN	Function	Returns the minimum value of a list of scalars	139
ODD	Function	Returns TRUE if the integer argument is odd	149
PRED	Function	Obtains the predecessor of an ordinal type	152
SIZEOF	Function	Returns the storage size of a variable or type	167
SUCC	Function	Obtains the successor of an ordinal type	171

Figure 120. Summary of Data Inquiry Routines

Data Movement Routines

Routine Name	Routine Type	Description	See Page
PACK	Procedure	Copies an array to a packed array	150
UNPACK	Procedure	Copies a packed array to an array	175

Figure 121. Summary of Data Movement Routines

General Routines

Routine Name	Routine Type	Description	See Page
TRACE	Procedure	Writes the routine return stack	173
HALT	Procedure	Stops program execution	128

Figure 122. Summary of General Routines

Input/Output Routines

Routine Name	Routine Type	Record or TEXT	Description	See Page
CLOSE	Procedure	Both	Closes a file	119
COLS	Function	TEXT	Returns the current column of the output line	119
EOF	Function	Both	Tests for end-of-file condition	124

Figure 123 (Part 1 of 2). Summary of Input/Output Routines

Routine Name	Routine Type	Record or TEXT	Description	See Page
EOLN	Function	TEXT	Tests for end-of-line condition	125
GET	Procedure	Both	Moves the file pointer to the next element of the input file	126
PAGE	Procedure	TEXT	Skips to the top of the next page	151
PDSIN	Procedure	Both	Opens a member of a partitioned data set for input	151
PDSOUT	Procedure	Both	Opens a member of a partitioned data set for output	152
PUT	Procedure	Both	Advances the file pointer to the next element of the output file	153
READ	Procedure	Both	Reads data from a file	154 155
READLN	Procedure	TEXT	Reads data from a file and advances the file pointer to the next line	155
RESET	Procedure	Both	Opens a file for input	163
REWRITE	Procedure	Both	Opens a file for output	163
SEEK	Procedure	Record	Positions an open file at a specific record	166
TERMIN	Procedure	TEXT	Opens a file for input from the terminal	171
TERMOUT	Procedure	TEXT	Opens a file for output to the terminal	171
UPDATE	Procedure	Record	Opens a file for input and output	176
WRITE	Procedure	Both	Writes data to a file	177 178
WRITELN	Procedure	TEXT	Writes data to a file and advances the file pointer to the next line	178

Figure 123 (Part 2 of 2). Summary of Input/Output Routines

Mathematical Routines

Routine Name	Routine Type	Description	See Page
ABS	Function	Computes the absolute value of a number	118
ARCTAN	Function	Returns the arctangent of the real argument	118
COS	Function	Returns the cosine of the real argument	121
EXP	Function	Returns the base of the natural log (e) raised to the power of the real argument	125
LN	Function	Returns the natural logarithm of the real argument	132
RANDOM	Function	Returns a pseudo-random number	154
SIN	Function	Returns the sine of the real argument	167
SQR	Function	Returns the square of a number	167

Figure 124 (Part 1 of 2). Summary of Mathematical Routines

Routine Name	Routine Type	Description	See Page
SQRT	Function	Returns the square root of the real argument	168

Figure 124 (Part 2 of 2). Summary of Mathematical Routines

Storage Management Routines

Routine Name	Routine Type	Description	See Page
DISPOSE	Procedure	Deallocates a dynamic variable	123
DISPOSEHEAP	Procedure	Frees a previously allocated heap	123
MARK	Procedure	Marks the beginning of a new subheap	136
NEW	Procedure	Allocates a dynamic variable from the current heap	143
NEWHEAP	Procedure	Creates a new heap	146
QUERYHEAP	Procedure	Identifies the current heap	153
RELEASE	Procedure	Deallocates one or more subheaps	162
USEHEAP	Procedure	Makes another heap the current heap	177

Figure 125. Summary of Storage Management Routines

String Routines for SBCS and DBCS Strings

Routine Name	Routine Type	Description	See Page
COMPRESS	Function	Replaces multiple blanks in a string with one blank	120
DELETE	Function	Returns a string with a portion removed	122
INDEX	Function	Finds the first occurrence of one string in another	130
LENGTH	Function	Returns the current length of a string	131
LPAD	Procedure	Pads or truncates a string on the left	133
LTOKEN	Procedure	Extracts tokens from a string	134
LTRIM	Function	Returns a string with leading blanks removed	135
MAXLENGTH	Function	Returns the maximum length of a string	137
READSTR	Procedure	Converts a string to values assigned to variables	160
RINDEX	Function	Finds the last occurrence of one string in another	164
RPAD	Procedure	Pads or truncates a string on the right	165
SUBSTR	Function	Returns a portion of a string	170
TOKEN	Procedure	Extracts tokens from a string	172
TRIM	Function	Returns a string with trailing blanks removed	174
WRITESTR	Procedure	Converts a series of expressions into a string	184

Figure 126. Summary of String Routines for SBCS and DBCS Strings

Note to Figure 126: These routines manipulate both SBCS and mixed strings in a byte-oriented manner. These routines manipulate DBCS strings in a double-byte oriented manner.

String Routines for Mixed Strings

Routine Name	Routine Type	Description	See Page
MCOMPRESS	Function	Replaces sequences of SBCS blanks with a single SBCS blank, and replaces sequences of DBCS blanks with a single DBCS blank	138
MDELETE	Function	Returns a mixed string with a portion removed	139
MINDEX	Function	Finds the first occurrence of one mixed string in another	140
MLENGTH	Function	Returns the length of a mixed string	140
MLTRIM	Function	Returns a mixed string with leading SBCS and DBCS blanks removed	141
MRINDEX	Function	Finds the last occurrence of one mixed string in another	141
MSUBSTR	Function	Returns a specific portion of a mixed string	142
MTRIM	Function	Returns a mixed string with trailing SBCS and DBCS blanks removed	143

Figure 127. Summary of String Routines for Mixed Strings

Note to Figure 127: These routines manipulate mixed strings in a character-oriented manner. Some of these routines return canonical mixed strings (strings with adjacent shift-out/shift-in pairs and adjacent shift-in/shift-out pairs removed).

System Access Routines

Routine Name	Routine Type	Description	See Page
CLOCK	Function	Returns the number of microseconds of execution	119
DATETIME	Procedure	Returns the current date and time of day	121
PARMS	Function	Returns the system dependent invocation parameters	151
RETCODE	Procedure	Sets the system dependent return code	163

Figure 128. Summary of System Access Routines

ABS Function

ABS returns the absolute value of its parameter, which can be any numeric type.

Figure 129 shows the definition of the ABS function.

```
FUNCTION ABS( i : INTEGER )
            : INTEGER;

FUNCTION ABS( r : REAL )
            : REAL;

FUNCTION ABS( s : SHORTREAL )
            : SHORTREAL;
```

Where	Represents
-------	------------

<i>i</i>	An integer expression
----------	-----------------------

<i>r</i>	A real expression
----------	-------------------

<i>s</i>	A shortreal expression
----------	------------------------

Figure 129. Definition of the ABS Function

ADDR Function

ADDR returns the location in storage of a given variable. Variables can be qualified variables, such as dereferenced pointers, subscripted variables, and fields of records.

Figure 130 shows the definition of the ADDR function.

```
FUNCTION ADDR( v : any-type)
            : INTEGER;

Where Represents
v      An identifier declared as a variable
```

Figure 130. Definition of the ADDR Function

ARCTAN Function

ARCTAN computes the arctangent of a floating-point number. The result is expressed in radians.

Figure 131 shows the definition of the ARCTAN function.

```
FUNCTION ARCTAN( x : REAL )
            : REAL;

Where Represents
x      An expression that evaluates to a real value
```

Figure 131. Definition of the ARCTAN Function

Real functions will accept integer and shortreal arguments. See "Type Compatibility" on page 46 for more information.

CHR Function

CHR returns the EBCDIC character corresponding to a given integer value. (Think of it as the inverse of ORD for characters.) Thus, ORD(CHR(I)) = I if I is in the subrange:

0..ORD(MAXCHAR)

If the operand is outside this range when checking is enabled, VS Pascal issues a run-time error message. If the operand is outside this range when checking is disabled, VS Pascal can act unpredictably.

Figure 132 shows the definition of the CHR function.

```
FUNCTION CHR( i : INTEGER )
           : CHAR;
```

Where Represents

i An integer expression that is to be interpreted as a character

Figure 132. Definition of the CHR Function

CLOCK Function

CLOCK returns the number of microseconds the program has been running.

Figure 133 shows the definition of the CLOCK function.

```
FUNCTION CLOCK : INTEGER;
```

Figure 133. Definition of the CLOCK Function

Note: In an MVS system, the time is "task" time; in a CMS system, the time is "CPU virtual" time.

CLOSE Procedure

CLOSE closes a specific file. You must reopen the file before you can use it again.

Figure 134 shows the definition of the CLOSE procedure.

```
PROCEDURE CLOSE( VAR f : filetype );
```

Where Represents

f A file variable

Figure 134. Definition of the CLOSE Procedure

COLS Function

COLS returns the current column number (position of the next character to be written) on the designated output file.

Figure 135 on page 120 shows the definition of the COLS function. COLS has no file name default.

```
FUNCTION COLS( CONST f : TEXT ) : INTEGER;
```

Where Represents

f A text file opened for output

Figure 135. Definition of the COLS Function

Note: You can force the output to a specific column with:

```
IF TAB > COLS(F) THEN
  WRITE(F, ' ':TAB-COLS(F));
```

COMPRESS Function

COMPRESS replaces sequences of SBCS blanks in an SBCS string with a single SBCS blank, and sequences of DBCS blanks in a DBCS string with a single DBCS blank.

Figure 136 shows the definition of the COMPRESS function.

```
FUNCTION COMPRESS( CONST source : STRING)
                  : STRING;
```

```
FUNCTION COMPRESS( CONST source : GSTRING)
                  : GSTRING;
```

Where Represents

source A string expression to be compressed

Figure 136. Definition of the COMPRESS Function

Note: Although COMPRESS is better suited to pure SBCS or DBCS strings, it can be used for mixed strings. COMPRESS manipulates mixed strings in a byte-oriented manner. However, MCOMPRESS is usually used for mixed strings.

Figure 137 shows examples of the COMPRESS function.

COMPRESS('A B CD ')	(* yields 'A B CD ' *)
COMPRESS('<.A.b> B CD ')	(* yields '<.Ab> B CD ' *)
COMPRESS('<.b.b.A.b.b>'G)	(* yields '<.b.A.b>'G *)

Figure 137. Examples of the COMPRESS Function

COS Function

COS computes the cosine of a floating-point number representing an angle in radians. Figure 138 shows the definition of the COS function.

```
FUNCTION COS( x : REAL )
           : REAL;
```

Where Represents

x An expression that evaluates to a real value

Figure 138. Definition of the COS Function

Real functions will accept integer and shortreal arguments. See "Type Compatibility" on page 46 for more information.

DATEIME Procedure

DATEIME returns the current date and time of day as two ALFA arrays. Figure 139 shows the definition of the DATEIME procedure.

```
PROCEDURE DATEIME( VAR date,time : ALFA );
```

Where Represents

date The returned date

time The returned time

Figure 139. Definition of the DATEIME Procedure

Figure 140 shows an example of the date and time format.

```
mm/dd/yy
hh:mi:ss
```

Where Represents

mm The month expressed as a two-digit value

dd The day of the month

yy The last two digits of the year

hh The hour of the day expressed in a 24-hour clock

mi The minute of the hour

ss The second of the minute

Figure 140. Example of the Date and Time Format

Note to Figure 140: At installation time, you can choose an alternate version of DATEIME that returns the date in this format:

```
dd/mm/yy
```

For information on how to install this alternative format, see *VS Pascal Installation and Customization for MVS*, or *VS Pascal Installation and Customization for VM*.

DELETE Function

DELETE returns a string with a specified portion removed.

Figure 141 shows the definition of the DELETE function.

```

FUNCTION DELETE( CONST source : STRING;
                  start  : INTEGER;
                  len    : INTEGER ) : STRING;

FUNCTION DELETE( CONST source : GSTRING;
                  start  : INTEGER;
                  len    : INTEGER ) : GSTRING;

```

Where Represents

source A string expression from which a portion will be deleted.

start An integer expression that specifies the starting position within the source where characters are to be deleted. The first character of the source string is at position 1.

len An optional integer expression that specifies the number of characters to be deleted. If *len* is omitted, it defaults to `LENGTH(s) - start + 1`; in other words, all remaining characters are deleted. (The string is truncated beginning at position *start*.)

Figure 141. Definition of the DELETE Function

Note: Although DELETE is better suited to pure SBCS or pure DBCS strings, it can be used for mixed strings. DELETE manipulates the strings in a byte-oriented manner. However, MDELETE is usually used for mixed strings.

Usage: To avoid an error message at run time:

- *start* must be greater than 0
- *len* must be greater than or equal to 0; if *len* is 0, the whole string is returned
- *start* + *len* - 1 must be less than or equal to the current length of the string.

Figure 142 shows an example of the DELETE function.

```

DELETE('ABCDE',2,3)      (* yields 'AE'      *)
DELETE('ABCDE',3)        (* yields 'AB'      *)
DELETE('ABCDE',3,1)      (* yields 'ABDE'   *)
DELETE('ABCDE',1)        (* yields ''        *)
DELETE('ABCDE',6,0)      (* yields 'ABCDE'  *)
DELETE('ABCDE',2,5)      (* yields an error *)

DELETE('<.A.B>CDE',2,2)    (* yields '<.B>CDE' *)
DELETE('AB<.C.D.E>',3)    (* yields 'AB'      *)

DELETE('<.A.B.C.D.E>'G,2,3) (* yields '<.A.E>'G *)

```

Figure 142. Example of the DELETE Function

DISPOSE Procedure

DISPOSE frees storage allocated for a single dynamic variable and, if the pointer is a variable, sets the pointer to NIL. DISPOSE always returns storage to the heap from which it was allocated. Figure 143 shows the definition of the DISPOSE procedure.

```

PROCEDURE DISPOSE(p1 : pointer );

PROCEDURE DISPOSE(p2 : pointer; t1,t2...: ordinal-type);

PROCEDURE DISPOSE(p3 : STRINGPTR;
                  len : INTEGER);

```

Where Represents

<i>p1</i>	A pointer expression returned from a call to NEW
<i>p2</i>	A pointer expression to a record returned from a call to NEW
<i>t1, t2</i>	Ordinal constants representing tag fields
<i>p3</i>	A string pointer expression returned from a call to NEW
<i>len</i>	An expression with an integer value

Figure 143. Definition of the DISPOSE Procedure

DISPOSE frees only the storage for a single dynamic variable; it does not recursively free any storage referenced by the dynamic variable (or any field of that dynamic variable). Thus, when you DISPOSE of an element of a linked list, you free storage only for that single element. If you intend to free storage for the entire list, you must DISPOSE every element in the list. It is your responsibility to ensure that a freed dynamic variable is not referenced by other pointers.

Note: It is an error to pass to DISPOSE a pointer that was not allocated by a call to NEW.

See "DISPOSEHEAP Procedure" for information on freeing an entire collection of dynamic variables within a separate heap. See "RELEASE Procedure" on page 162 for information on freeing an entire subheap.

See Figure 203 on page 148 to see how DISPOSE is used in conjunction with NEWHEAP and DISPOSEHEAP.

DISPOSEHEAP Procedure

DISPOSEHEAP deallocates a heap created by the NEWHEAP routine. DISPOSEHEAP returns all storage associated with the heap to either the VS Pascal run-time environment or to the operating system, depending upon the DISP option specified when the heap was created with NEWHEAP. Deallocating a heap with DISPOSEHEAP frees all dynamic variables and subheaps contained within that heap.

Figure 144 shows the definition of the DISPOSEHEAP procedure.

```

PROCEDURE DISPOSEHEAP( VAR p : pointer );

```

Where Represents

<i>p</i>	A pointer returned from a call to NEWHEAP
----------	---

Figure 144. Definition of the DISPOSEHEAP Procedure

After you deallocate a heap with DISPOSEHEAP, the heap-id is set to NIL. If you deallocate the current heap, a current heap no longer exists; an error will not be returned.

Note: It is an error to pass to DISPOSEHEAP a pointer that was not allocated by a call to NEWHEAP.

See Figure 203 on page 148 to see how DISPOSEHEAP is used in conjunction with NEWHEAP, QUERYHEAP, and USEHEAP.

EOF Function

EOF tests a file for the end-of-file condition. EOF returns TRUE when the end-of-file condition is true for the file; otherwise, it returns FALSE. The EOF condition occurs on any attempt to read an input file past the last record element of the file. EOF also returns TRUE when the file is open for output. Figure 145 shows the definition of the EOF function.

```
FUNCTION EOF( f : filetype ) : BOOLEAN;
```

Where Represents

f An optional file variable; the default is the predefined file INPUT

Figure 145. Definition of the EOF Function

Figure 146 shows an example of testing for end-of-file condition. All of the records are read from SYSIN and written to SYSOUT.

```
TYPE
  FREC = RECORD
    A,B : INTEGER
  END;

VAR
  SYSIN,
  SYSOUT: FILE OF FREC;

BEGIN
  RESET(SYSIN);
  REWRITE(SYSOUT);
  WHILE NOT EOF(SYSIN) DO
  BEGIN
    SYSOUT@ := SYSIN@;
    PUT(SYSOUT);
    GET(SYSIN);
  END;
END;
```

Figure 146. Example of Testing for End-of-File Condition

EOLN Function

EOLN tests a text file for the end-of-line condition. EOLN returns TRUE if the file is positioned at an end-of-line; otherwise, it returns FALSE. Figure 147 shows the definition of the EOLN function.

```
FUNCTION EOLN( f : TEXT ) : BOOLEAN;
```

Where Represents

f An optional file variable of type TEXT; the default is the predefined file INPUT

Figure 147. Definition of the EOLN Function

If EOLN is TRUE, the file pointer will point to a blank. Although the blank is not in the file, it appears as if it were. Usually, the extra blank will not affect your results, but if the physical layout of the data is important, use EOLN with care.

Figure 148 shows an example of copying a text file. The file is copied from SYSIN to SYSOUT.

```
VAR
  SYSIN,
  SYSOUT : TEXT;

BEGIN
  RESET(SYSIN);
  REWRITE(SYSOUT);
  WHILE NOT EOF(SYSIN) DO
    BEGIN
      WHILE NOT EOLN(SYSIN) DO
        BEGIN
          SYSOUT@ := SYSIN@;
          PUT(SYSOUT);
          GET(SYSIN);
        END;
      WRITELN(SYSOUT);
      READLN(SYSIN);
    END;
  END;
```

Figure 148. Example of Copying a Text File

EXP Function

EXP computes the value of the base of the natural logarithm, *e*, raised to the power expressed by a floating-point number.

Figure 149 shows the definition of the EXP function.

```
FUNCTION EXP( x : REAL )
  : REAL;
```

Where Represents

x An expression that evaluates to a real value

Figure 149. Definition of the EXP Function

Real functions will accept integer **and shortreal** arguments. See "Type Compatibility" on page 46 for more information.

FLOAT Function

FLOAT converts an integer value to a floating-point value. Use FLOAT when you need to make this conversion explicit in the program. VS Pascal implicitly converts an integer to a real value when one operand of an arithmetic or relational operator is a real type and the other is an integer type. Conversion is also done on parameter passing. See "Type Compatibility" on page 46 for more information.

Figure 150 shows the definition of the FLOAT function.

```
FUNCTION FLOAT( i : INTEGER )
               : REAL;
```

Where Represents

i An expression that has an integer value

Figure 150. Definition of the FLOAT Function

GET Procedure

GET positions the file pointer of a file (previously opened for input) to the next component in the file. For example, if the file is defined as a FILE OF INTEGER, each GET returns the next INTEGER. A GET on a file of type TEXT returns a single character. Figure 151 shows the definition of the GET procedure.

```
PROCEDURE GET( f : filetype );
```

Where Represents

f A file variable

Figure 151. Definition of the GET Procedure

Restriction: GET cannot read pure DBCS data from a text file.

GSTR Function

GSTR converts a GCHAR or a DBCS fixed string to a GSTRING. GSTR can also be applied to a GSTRING, but no operation is performed. Figure 152 on page 127 shows the definition of the GSTR function.

```

FUNCTION GSTR( x : GCHAR)
    : GSTRING;

FUNCTION GSTR( x : PACKED ARRAY[1..n] OF GCHAR )
    : GSTRING;

FUNCTION GSTR( x : GSTRING)
    : GSTRING;

```

Where Represents

x A DBCS character, DBCS fixed string, or DBCS string expression

Figure 152. Definition of the GSTR Function

Figure 153 shows an example of the GSTR function.

```

VAR
    GC : GCHAR;
    GA : PACKED ARRAY[1..4] OF GCHAR;
    G4 : GSTRING(4);

BEGIN
    GC := '<.A>'G;      (* .A is stored in GC *)
    G4 := GSTR(GC);    (* .A is stored in G4 *)
    GA := '<.A.B>'G;    (* .A.B is stored in GA *)
    G4 := GSTR(GA);    (* .A.B is stored in G4 *)
END;

```

Figure 153. Example of the GSTR Function

GTOSTR Function

GTOSTR converts a GSTRING to a STRING, adding a shift-out character at the beginning of the STRING and a shift-in character at the end of the STRING.

Figure 154 shows the definition of the GTOSTR function.

```

FUNCTION GTOSTR( x : GSTRING)
    : STRING;

```

Where Represents

x A DBCS string expression

Figure 154. Definition of the GTOSTR Function

Figure 155 on page 128 shows an example of the GTOSTR function.

```

VAR
  G : GSTRING(4);
  S : STRING(10);

BEGIN
  G := <.A.B>'G;      (* .A.B is stored in G *)
  S := GTOSTR(G);     (* <.A.B> is stored in S *)
END;
```

Figure 155. Example of the GTOSTR Function

HALT Procedure

HALT stops execution of a VS Pascal program. Consider it a return from the main program. Figure 156 shows the definition of the HALT procedure.

```
PROCEDURE HALT;
```

Figure 156. Definition of the HALT Procedure

HBOUND Function

HBOUND returns the upper bound of an array's index. You can specify the array in two ways:

- As an identifier declared as an array via the type construct
- As a variable that is of type ARRAY.

The type of the value returned is the same as the type of the index. The second parameter defines the dimension of the array for which the upper bound is returned.

HBOUND also works on SPACE types.

Figure 157 shows the definition of the HBOUND function.

```

FUNCTION HBOUND( a : array-type;
                 i : integer-const)
               : ordinal-type;
```

Where Represents

a	An identifier declared as an array type or variable
i	An optional constant expression that has a positive integer value; the default is 1

Figure 157. Definition of the HBOUND Function

Figure 158 on page 129 shows an example of the HBOUND function.

```

TYPE
  GRID = ARRAY[-10..10,-5..5] OF REAL;

VAR
  A : GRID;
  B : ARRAY[ 1..100 ] OF
      ARRAY[ 0..9 ] OF CHAR;
  .
  HBOUND( A )                (* IS 10 *)
  HBOUND( GRID )              (* IS 10 *)
  HBOUND( B, 2 )              (* IS 9 *)
  HBOUND( B[1] )              (* IS 9 *)

```

Figure 158. Example of the HBOUND Function

HIGHEST Function

HIGHEST returns the highest value that can be represented by the operand. The operand can be a type identifier or a variable. If the operand is a type identifier, HIGHEST returns the highest value that can be assigned to a variable of that type. If the operand is a variable, HIGHEST returns the highest value that can be assigned to that variable. Figure 159 shows the definition of the HIGHEST function.

```

FUNCTION HIGHEST( s : ordinal-type)
  : ordinal-type;

```

Where Represents

s An identifier declared as an ordinal type or variable

Figure 159. Definition of the HIGHEST Function

Figure 160 shows an example of the HIGHEST function.

```

TYPE
  DAYS = (SUN, MON, TUE, WED,
          THU, FRI, SAT);
  SMALL = 0 .. 31;

VAR
  I : INTEGER;
  J : 0 .. 255;
  .
  HIGHEST(DAYS)  (* IS SAT *)
  HIGHEST(BOOLEAN) (* IS TRUE *)
  HIGHEST(SMALL) (* IS 31 *)
  HIGHEST(I)     (* IS MAXINT *)
  HIGHEST(J)     (* IS 255 *)

```

Figure 160. Example of the HIGHEST Function

INDEX Function

INDEX returns the position of the first occurrence of the second string within the first. If the second string does not exist in the first string, INDEX returns a zero. If the second string is null, a 1 will always be returned.

Figure 161 shows the definition of the INDEX function.

```

FUNCTION INDEX( CONST source : STRING;
                CONST lookup : STRING )
                : 0..32767;

FUNCTION INDEX( CONST source : GSTRING;
                CONST lookup : GSTRING )
                : 0..16382;
```

Where Represents

source A string expression to which *lookup* is compared

lookup A string expression which will be compared to *source*

Figure 161. Definition of the INDEX Function

Note: Although INDEX is better suited to pure SBCS or pure DBCS strings, it can be used for mixed strings. INDEX handles the strings in a byte-oriented manner. However, MINDEX is usually used for mixed strings.

Figure 162 shows an example of the INDEX function.

```

VAR
  INDEX('ABCABCABC','BC')           (* yields 2 *)
  INDEX('ABCABCABC','X')             (* yields 0 *)
  INDEX('<.A.B>CABCABC','<.A.B>C')     (* yields 1 *)
  INDEX('<.A.B.C>'G','<.B>G')          (* yields 2 *)
```

Figure 162. Example of the INDEX Function

LBOUND Function

LBOUND returns the lower bound of an array's index. The array can be specified in two ways:

- As an identifier declared as an array via the type construct
- As a variable that is of type ARRAY.

The type of the value returned is the same as the type of the index. The second parameter defines the dimension of the array for which the lower bound is returned.

LBOUND also works on SPACE types.

Figure 163 on page 131 shows the definition of the LBOUND function.

```

FUNCTION LBOUND( a : array-type;
                 i : integer-const)
                 : ordinal-type;

```

Where Represents

a An identifier declared as an array type or variable

i An optional constant expression that has a positive integer value; the default is 1

Figure 163. Definition of the LBOUND Function

Figure 164 shows an example of the LBOUND function.

```

TYPE
  GRID = ARRAY[-10..10,-5..5] OF REAL;
VAR
  A : GRID;
  B : ARRAY[ 1..100 ] OF
      ARRAY[ 0..9 ] OF CHAR;
.
.
LBOUND( A )           (* IS -10 *)
LBOUND( GRID )        (* IS -10 *)
LBOUND( B, 2 )        (* IS 0  *)
LBOUND( B[1] )        (* IS 0  *)

```

Figure 164. Example of the LBOUND Function

LENGTH Function

LENGTH returns the current length of a specified string. For SBCS strings, the value will be in the range 0 to 32767. For DBCS strings, the value will be in the range of 0 to 16382 characters.

Figure 165 shows the definition of the LENGTH function.

```

FUNCTION LENGTH( s : STRING )
               : 0..32767;

FUNCTION LENGTH( s : GSTRING )
               : 0..16382;

```

Where Represents

s An expression with a string value

Figure 165. Definition of the LENGTH Function

Note: Although LENGTH is better suited to pure SBCS or pure DBCS strings, it can be used for mixed strings. LENGTH handles the strings in a byte-oriented manner. However, MLENGTH is usually used for mixed strings.

LOWEST Function

Figure 166 shows examples of the LENGTH function.

LENGTH('ABCD')	(* yields 4 *)
LENGTH('<.A.B.C.D>'G)	(* yields 4 *)

Figure 166. Examples of the LENGTH Function

LN Function

LN computes the natural logarithm of a floating-point number. Figure 167 shows the definition of the LN function.

```
FUNCTION LN( x : REAL )
           : REAL;
```

Where Represents

x An expression that evaluates to a real value

Figure 167. Definition of the LN Function

Real functions will accept integer and shortreal arguments. See "Type Compatibility" on page 46 for more information.

LOWEST Function

LOWEST returns the lowest value that can be represented by the operand. The operand can be a type identifier or a variable. If the operand is a type identifier, LOWEST returns the lowest value that can be assigned to a variable of that type. If the operand is a variable, LOWEST returns the lowest value that can be assigned to that variable. Figure 168 shows the definition of the LOWEST function.

```
FUNCTION LOWEST( s : ordinal-type)
               : ordinal-type;
```

Where Represents

s An identifier declared as an ordinal type or variable

Figure 168. Definition of the LOWEST Function

Figure 169 shows an example of the LOWEST function.

```

TYPE
  DAYS  = (SUN, MON, TUE, WED,
           THU, FRI, SAT);
  SMALL = 0 .. 31;

VAR
  I      : INTEGER;
  J      : 0 .. 255;
  .
  .
  .
  LOWEST(DAYS)      (* IS SUN  *)
  LOWEST(BOOLEAN)   (* IS FALSE *)
  LOWEST(SMALL)      (* IS 0    *)
  LOWEST(I)          (* IS MININT *)
  LOWEST(J)          (* IS 0    *)
  
```

Figure 169. Example of the LOWEST Function

LPAD Procedure

LPAD pads or truncates a string on the left. LPAD manipulates pure SBCS or pure DBCS strings.

Figure 170 shows the definition of the LPAD procedure.

```

PROCEDURE LPAD( VAR s : STRING;
                l : INTEGER;
                c : CHAR);

PROCEDURE LPAD( VAR s : GSTRING;
                l : INTEGER;
                c : GCHAR);
  
```

Where	Represents
s	The string to be padded
l	The final length of s
c	The pad character

Figure 170. Definition of the LPAD Procedure

Note to Figure 170: If LENGTH(s) is greater than l, then the characters are truncated on the left. If LENGTH(s) is less than l, then s is extended with the character c on the left.

Figure 171 shows examples of the LPAD procedure.

```

S := 'ABCDEF';
LPAD(S, 10, '$');      (* yields '$$$$ABCDEF' in S *)

S := 'ABCDEF';
LPAD(S, 5, '$');       (* yields 'BCDEF' in S      *)

G := '<.A.B.C.D.E.F>'G;
LPAD(G, 10, '<.$>'G);   (* yields '<.$.$.$A.B.C.D.E.F>'G in G *)

G := '<.A.B.C.D.E.F>'G;
LPAD(G, 5, '<.$>'G);    (* yields '<.B.C.D.E.F>'G in G      *)

```

Figure 171. Examples of the LPAD Procedure

Note: Because LPAD is now a predefined routine, %INCLUDE STRING is no longer required to invoke the LPAD function.

LTOKEN Procedure

Starting from a given position, LTOKEN scans a string for a token, and returns the token in a string. Figure 172 shows the definition of the LTOKEN procedure.

```

PROCEDURE LTOKEN( VAR pos   : INTEGER;
                  CONST source : STRING;
                  VAR result : STRING);

```

Where Represents

pos An integer corresponding to the position in the source string where the search for the token begins. The value of this integer gets updated to reflect the starting position for subsequent calls to LTOKEN.

source A string expression that contains the data from which a token is to be extracted.

result The resulting token.

Figure 172. Definition of the LTOKEN Procedure

When LTOKEN scans a string, it ignores leading, multiple, and trailing blanks. If there is no token in the string, the value of the first parameter *pos* is set to $\text{LENGTH}(\text{source}) + 1$, and the *result* parameter is set to one blank. If the token is longer than the result variable, an error will result.

A token can be any of the following:

- A VS Pascal identifier, any number of alphanumeric characters, "\$", or an underscore. The first letter must be alphabetic or a "\$".
- A VS Pascal unsigned integer. (See "Types of Constants" on page 36.)
- The following special symbols:

+	-	*	/	->	@	¢
=	<>	<	<=	>=	>	!
()	[]	'	"	%
	&	&&		~	=	#
:	:	:=	.	,	..	><
{	}	(*	*)	/*	*/	
(.	.)	<<	>>			

- Any other single character not listed above.

Figure 173 shows an example of the LTOKEN procedure.

```

I := 2;
LTOKEN(I, ' ', Token+, RESULT)  (* I is set to 8          *)
                                (* RESULT is set to 'Token' *)

```

Figure 173. Example of the LTOKEN Procedure

Note to Figure 173: LTOKEN would return the same if I were set to 3; that is, leading blanks are ignored.

LTRIM Function

LTRIM returns the specified parameter's value with all leading blanks removed.

Figure 174 shows the definition of the LTRIM function.

```

FUNCTION LTRIM( CONST source : STRING )
                : STRING;

FUNCTION LTRIM( CONST source : GSTRING )
                : GSTRING;

```

Where Represents

source A string expression

Figure 174. Definition of the LTRIM Function

Note: Although LTRIM is better suited to pure SBCS and pure DBCS strings, it can be used for mixed strings. LTRIM manipulates mixed strings in a byte-oriented manner. However, MLTRIM is usually used for mixed strings.

Figure 175 shows examples of the LTRIM function.

LTRIM(' A B ')	(* yields 'A B ' *)
LTRIM(' ')	(* yields ' ' *)
LTRIM(' <.A> B ')	(* yields '<.A> B ' *)
LTRIM('<.b.b.b.A>'G)	(* yields '<.A>'G *)

Figure 175. Examples of the LTRIM Function

MARK Procedure

MARK creates a logical "bookmark" in the current heap, allowing all subsequently allocated dynamic variables in that heap to be deallocated quickly with a single call to the RELEASE procedure.

Figure 176 shows the definition of the MARK procedure.

```
PROCEDURE MARK( VAR p : pointer );
```

Where Represents
p A pointer to any type

Figure 176. Definition of the MARK Procedure

In this simple example,

```
MARK(X)
  NEW(A)
  NEW(B)
MARK(Y)
  NEW(C)
```

the call RELEASE(Y) would deallocate only the dynamic variable pointed to by C. The call RELEASE(X) would free the dynamic variables pointed to by A, B, and C.

The pointer passed to MARK is called a *subheap pointer*. It must not be used as the base of a dynamic variable or unpredictable results can occur.

Figure 177 on page 137 shows an example of using MARK and RELEASE within a single heap. See Figure 203 on page 148 for an example using multiple heaps.

```

TYPE
  MARKP = @INTEGER;
  LINKP = @LINK;
  LINK = RECORD
    NAME: STRING(30);
    NEXT: LINKP
  END;
VAR
  P      : MARKP;
  Q1,
  Q2,
  Q3     : LINKP;
BEGIN
  .
  .
  MARK(P);
  .
  .
  NEW(Q1);
  NEW(Q2);
  NEW(Q3);
  .
  .
  RELEASE(P);          (* Frees Q1, Q2 and Q3    *)
  .
  .
END;

```

Figure 177. Example of Using MARK and RELEASE within a Single Heap

MAX Function

MAX returns the maximum value of one or more expressions.

The parameters for MAX can be a mixture of integer, real, and shortreal parameters. If the parameters are mixed and one of them is a real, a real value will be returned. If the parameters are mixed and do not include a real, a shortreal value will be returned. Otherwise, MAX will return a value compatible with the types of the parameters. Figure 178 shows the definition of the MAX function.

```

FUNCTION MAX( expr,expr.. : scalar-type)
              : scalar-type;

```

Where Represents

expr A scalar expression, including DBCS character, real, and shortreal

Figure 178. Definition of the MAX Function

MAXLENGTH Function

MAXLENGTH returns the maximum length of a specified string. It works on either STRING or GSTRING data types. For SBCS strings, the value will be in the range 0 to 32767. For DBCS strings, the value will be in the range 0 to 16382.

Figure 179 on page 138 shows the definition of the MAXLENGTH function.

```
FUNCTION MAXLENGTH( s : STRING )
    : 0..32767;

FUNCTION MAXLENGTH( s : GSTRING )
    : 0..16382;
```

Where Represents

s An expression with a string value

Figure 179. Definition of the MAXLENGTH Function

Figure 180 shows an example of the MAXLENGTH function.

```
VAR
    S : STRING (8);
    G : GSTRING (4);
BEGIN
    MAXLENGTH (S);           (* yields 8 *)
    MAXLENGTH (G);           (* yields 4 *)
END;
```

Figure 180. Example of the MAXLENGTH Function

MCOMPRESS Function

MCOMPRESS returns a string with sequences of SBCS blanks replaced with a single SBCS blank, and sequences of DBCS blanks replaced with a single DBCS blank. Figure 181 shows the definition of the MCOMPRESS function.

```
FUNCTION MCOMPRESS(CONST msource : STRING)
    : STRING;
```

Where Represents

msource A mixed string expression to be compressed

Figure 181. Definition of the MCOMPRESS Function

MCOMPRESS manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct. It returns a canonical mixed string.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when COMPRESS is used on pure SBCS strings.

Figure 182 shows an example of the MCOMPRESS function.

```
S := 'bb<.b.b.b.b>bbbb<.b.b.b><.b.b>bb'
MCOMPRESS(S)           (* yields 'b<.b>b<.b>b' *)
```

Figure 182. Example of the MCOMPRESS Function

MDELETE Function

MDELETE returns a mixed string with a specified portion removed. Figure 183 shows the definition of the MDELETE function.

```
FUNCTION MDELETE( CONST msource : STRING;
                  start   : INTEGER;
                  len     : INTEGER ) : STRING;
```

Where Represents

msource A mixed string expression from which a portion will be deleted.

start An integer expression that specifies the starting position within the source where characters are to be deleted. The first character of the source string is at position 1.

len An optional integer expression that specifies the number of characters to be deleted. If *len* is omitted, it defaults to `MLENGTH(s) - start + 1`; in other words, all remaining characters are deleted. (The string is truncated beginning at position *start*.)

Figure 183. Definition of the MDELETE Function

Usage: To avoid an error message at run time:

- *start* must be greater than 0.
- *len* must be greater than or equal to 0. If *len* is 0, the whole string is returned.
- *start* + *len* - 1 must be less than or equal to the current character length of the string.

MDELETE manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct. It returns a canonical mixed string.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when DELETE is used on pure SBCS strings.

Figure 184 shows an example of the MDELETE function.

```
MDELETE('a<.B.C>d',1,2)      (* yields '<.C>d' *)
MDELETE('<.A.B><.C>',2,1)      (* yields '<.A.C>' ; adjacent SI/SO pair removed *)
MDELETE('<.A.B><<.C>',2,1)     (* yields '<.A.C>' ; adjacent SO/SI pairs removed *)
```

Figure 184. Example of the MDELETE Function

MIN Function

MIN returns the minimum value of one or more expressions.

The parameters for MIN can be a mixture of integer, real, and shortreal expressions. If the parameters are mixed and one of them is a real, a real value will be returned. If the parameters are mixed and do not include a real, a shortreal value will be returned. Otherwise, MIN will return a value compatible with the types of the parameters.

Figure 185 on page 140 shows the definition of the MIN function.

```

FUNCTION MIN( expr,expr.. : scalar-type)
                : scalar-type;

```

Where Represents

expr A scalar expression, including DBCS character, real, and shortreal

Figure 185. Definition of the MIN Function

MINDEX Function

MINDEX returns the position of the first occurrence of the second mixed string within the first mixed string. If the second mixed string does not exist in the first mixed string, MINDEX returns a zero. If the second mixed string is null, a 1 will always be returned. Figure 186 shows the definition of the MINDEX function.

```

FUNCTION MINDEX( CONST msource : STRING;
                  CONST mlookup : STRING )
                : 0..32767;

```

Where Represents

msource A mixed string expression to which *mlookup* is compared

mlookup A mixed string expression which will be compared to *msource*

Figure 186. Definition of the MINDEX Function

MINDEX manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when INDEX is used on pure SBCS strings.

Figure 187 shows an example of the MINDEX function.

```

MINDEX('<.A.B>','<.A>')    (* yields 1 *)
MINDEX('<.A.B>','A')      (* yields 0 because an SBCS 'A' *)
                          (* is not the same as a DBCS 'A' *)

```

Figure 187. Example of the MINDEX Function

MLENGTH Function

MLENGTH returns the character length of a mixed string. Figure 188 shows the definition of the MLENGTH function.

```

FUNCTION MLENGTH(m : string)
                : 0..32767;

```

Where Represents

m A mixed string expression

Figure 188. Definition of the MLENGTH Function

MLENGTH manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when LENGTH is used on pure SBCS strings.

Figure 189 shows an example of the MLENGTH function.

```
MLENGTH('a<.B.C>d')      (* yields 4 *)
```

Figure 189. Example of the MLENGTH Function

MLTRIM Function

MLTRIM returns a string with leading SBCS and DBCS blanks removed.

Figure 190 shows the definition of the MLTRIM function.

```
FUNCTION MLTRIM( CONST msource : STRING )
                                : STRING;
```

Where Represents

msource A mixed string expression

Figure 190. Definition of the MLTRIM Function

MLTRIM manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when LTRIM is used on pure SBCS strings.

Figure 191 shows an example of the MLTRIM function.

```
MLTRIM('b<.b.b>b')          (* yields ' ' *)
MLTRIM('b<.b.b>AB<.C>')      (* yields 'AB<.C>' *)
```

Figure 191. Example of the MLTRIM Function

MRINDEX Function

MRINDEX returns the position of the last occurrence of the second mixed string within the first mixed string. If the second mixed string does not exist in the first mixed string, MRINDEX returns a zero. If the second mixed string is null, MLENGTH(s) + 1 will always be returned. Figure 192 on page 142 shows the definition of the MRINDEX function.

```

FUNCTION MRINDEX( CONST msource : STRING;
                  CONST mlookup : STRING )
                  : 0..32768;

```

Where Represents

msource A mixed string expression to which *mlookup* is compared

mlookup A mixed string expression which will be compared to *msource*

Figure 192. Definition of the MRINDEX Function

MRINDEX manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when RINDEX is used on pure SBCS strings.

Figure 193 shows an example of the MRINDEX function.

```

MRINDEX('<.A.B.C.A>', '<.A>')      (* yields 4                *)
MRINDEX('<.A.B.C.A>', 'A')        (* yields 0, because an SBCS 'A' *)
                                (* is not the same as a DBCS 'A' *)

```

Figure 193. Example of the MRINDEX Function

MSUBSTR Function

MSUBSTR returns a specified portion of a mixed string. Figure 194 shows the definition of the MSUBSTR function.

```

FUNCTION MSUBSTR( CONST msource : STRING;
                  start   : INTEGER;
                  len     : INTEGER) : STRING;

```

Where Represents

msource A mixed string expression from which a substring will be returned.

start An integer expression that specifies the starting position within the source from which the substring is to be extracted. The first character of the source string is at position 1.

len An optional integer expression that determines the length of the substring. If *len* is omitted, it defaults to $\text{MSUBSTR}(s) - \text{start} + 1$; in other words, the substring returned will be the remaining portion of the source string from position *start*.

Figure 194. Definition of the MSUBSTR Function

Usage: To avoid an error message at run time, be sure that:

- *start* must be greater than 0.
- *len* must be greater than or equal to 0. If *len* is 0, a null string is returned.
- $\text{start} + \text{len} - 1$ must be less than or equal to the character length of the string.

MSUBSTR manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct. It returns a canonical mixed string.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when SUBSTR is used on pure SBCS strings.

Figure 195 shows an example of the MSUBSTR function.

MSUBSTR('a<.B.C>d',2,3)	(* yields '<.B.C>d' *)
MSUBSTR('a<.B.C>d',1,3)	(* yields 'a<.B.C>' *)
MSUBSTR('a<.B.C>d',3)	(* yields '<.C>d' *)
MSUBSTR('a<.B.C>d',1)	(* yields 'a<.B.C>d' *)
MSUBSTR('a<.B.C>d',5,0)	(* yields '' *)
MSUBSTR('a<.B.C>d',2,5)	(* is an error *)

Figure 195. Example of the MSUBSTR Function

MTRIM Function

MTRIM returns a string with trailing SBCS and DBCS blanks removed.

Figure 196 shows the definition of the MTRIM function.

```
FUNCTION MTRIM( CONST msource : STRING )
                : STRING;
```

Where Represents

msource A mixed string expression

Figure 196. Definition of the MTRIM Function

MTRIM manipulates mixed strings in a character-oriented manner, treating SBCS characters and DBCS characters as distinct.

Note: If the mixed string contains no DBCS portions, the results are the same as those obtained when TRIM is used on pure SBCS strings.

Figure 197 shows an example of the MTRIM function.

MTRIM('b<.b.b>b')	(* yields '' *)
MTRIM('AB<.C>b<.b.b>')	(* yields 'AB<.C>' *)

Figure 197. Example of the MTRIM Function

NEW Procedure

NEW allocates storage for a dynamic variable from the current heap and sets the pointer to point to the dynamic variable. Figure 198 on page 144 shows the definition of the NEW procedure.

Form 1:

```
PROCEDURE NEW(VAR p1 : pointer );
```

Form 2:

```
PROCEDURE NEW(VAR p2 : pointer;
               t1,t2... : ordinal-type);
```

Form 3:

```
PROCEDURE NEW(VAR p3 : STRINGPTR;
               len : INTEGER);
```

Where Represents

<i>p1</i>	A pointer
<i>p2</i>	A pointer to a record that contains a variant part
<i>t1, t2</i>	Ordinal constants representing tag fields
<i>p3</i>	A string pointer
<i>len</i>	An expression with an integer value

Figure 198. Definition of the NEW Procedure

Form 1 allocates the storage necessary to represent a value of the type to which the pointer refers. If the type of the dynamic variable is a record with a variant part, the space allocated is the amount required for the record when the largest variant is active. Figure 199 shows an example of the NEW procedure (form 1).

```
TYPE
  LINKP = @LINK;
  LINK = RECORD
    NAME : STRING(30);
    NEXT : LINKP;
  END;

VAR
  P,
  HEAD : LINKP;
  .
  .
BEGIN
  .
  .
  NEW(P);
  WITH P@ DO
    BEGIN
      NAME := 'MIKI WALTER';
      NEXT := HEAD;
    END;
  HEAD := P;
  .
  .
END;
```

Figure 199. Example of the NEW Procedure (Form 1)

Form 2 allocates a variant record when it is known which variants will be active. The amount of storage allocated is no larger than necessary to contain the specified variants. The ordinal constants are tag field values. The first one indicates which variant in the record is active; the second one represents which variant in the first variant is active, and so on.

Note: The NEW procedure *does not set tag fields*. The tag list serves only to indicate the amount of storage required; it is your responsibility to set the tag fields after the record is allocated.

Figure 200 shows an example of the NEW Procedure (form 2).

```

TYPE
  AGE = 0..100;
  RECP = @REC;
  REC =
    RECORD
      NAME: STRING(30);
      CASE HOWOLD: AGE OF
        0..18:
          (FATHER: RECP);
        19..100:
          (CASE MARRIED: BOOLEAN OF
            TRUE: (SPOUSE: RECP);
            FALSE: ()
          );
      END;
  VAR
    P : RECP;
  .
  .
  BEGIN
    .
    .
    NEW(P,18);
    WITH P@ DO
      BEGIN
        NAME := 'PAUL FEHDER, JR';
        HOWOLD := 18;
        NEW(FATHER,54,TRUE);
        WITH FATHER@ DO
          BEGIN
            NAME := 'PAUL FEHDER';
            HOWOLD := 54;
            MARRIED := TRUE;
            NEW(SPOUSE,50,TRUE);
            .
            .
          END (*with father@*);
        END (*with p@*);
        .
        .
      END;
    .
  END;

```

Figure 200. Example of the NEW Procedure (Form 2)

Form 3 allocates a string whose maximum length is known only during program execution. The amount of storage to be allocated for the string is defined by the required second parameter.

Figure 201 shows an example of the NEW procedure (form 3).

```
VAR
P      : STRINGPTR;
Q      : STRINGPTR;
I      : 0..32767;
BEGIN
.
.
I := 30;
NEW(P, I);
WRITELN( MAXLENGTH(P@) );
      (*writes '30' to output*)
NEW(Q,5);
Q@ := '1234567890';
      (*causes a truncation*)
      (*error at execution *)
END;
```

Figure 201. Example of the NEW Procedure (Form 3)

NEWHEAP Procedure

NEWHEAP reserves a block of storage in the VS Pascal run-time environment from which dynamic variables can be allocated. This block of storage is called a *heap*, and is referred to by the heap-id returned from NEWHEAP. Characteristics of a heap include initial size, extension size, location, and disposition. These characteristics can be specified with the NEWHEAP procedure. Figure 202 on page 147 shows the definition of the NEWHEAP procedure.

```
PROCEDURE NEWHEAP(VAR p : pointer; CONST s : string);
```

Where	Represents
<i>p</i>	A pointer to any type
<i>s</i>	An optional string containing any combination of the following options, separated by commas:

INIT = *nnn*

Specifies the number of kilobytes initially allocated for the new heap, where *nnn* is a positive integer. VS Pascal always uses this value regardless of any initial heap size you might specify on the HEAP run-time option. If you do not specify an initial size on NEWHEAP, VS Pascal defaults to the initial size you specify on HEAP. If you do not specify an initial size on either NEWHEAP or HEAP, VS Pascal defaults to 12 kilobytes.

INCR = *nnn*

Specifies the number of kilobytes the heap is extended if necessary, where *nnn* is a positive integer. If you do not specify an increment size on NEWHEAP, VS Pascal defaults to the increment size you specify on the HEAP run-time option. If you do not specify an increment size on either NEWHEAP or the HEAP run-time option, VS Pascal defaults to 12 kilobytes.

LOC = ANY|BELOW

Specifies where the heap is located in memory.

- ANY lets VS Pascal allocate the heap anywhere in storage.
- BELOW forces the heap into the 24-bit address space.

Default: ANY.

DISP = KEEP|FREE

Specifies how the heap is disposed.

- KEEP lets VS Pascal retain the memory space for possible future use. A DISPOSEHEAP on such a heap returns the memory to the VS Pascal environment, where it can be used by another heap with similar characteristics.
- FREE forces VS Pascal to immediately return storage to the operating system. A DISPOSEHEAP on such a heap issues an immediate FREEMAIN.

Default: KEEP.

Figure 202. Definition of the NEWHEAP Procedure

The pointer passed to NEWHEAP is called a *heap-id*. It must not be used as the base of a dynamic variable or unpredictable results might occur.

Figure 203 shows an example of the NEWHEAP procedure.

```

PROGRAM MULTIPLE_HEAPS;

TYPE
  DUMMY_TYPE = INTEGER;

  HEAP_POINTER = @DUMMY_TYPE;      (* This creates an      *)
                                   (* identifier type for    *)
  MARK_POINTER = @DUMMY_TYPE;      (* heaps and for marks.  *)

  PHRASE_POINTER = @PHRASE;
  PHRASE = STRING(30);

VAR
  DEFAULT_HEAP,
  FIRST_HEAP: HEAP_POINTER;

  MARK_ONE: MARK_POINTER;

  PHRASE_A,
  PHRASE_B,
  PHRASE_C,
  PHRASE_D: PHRASE_POINTER;

BEGIN
  QUERYHEAP(DEFAULT_HEAP);          (* Saves the heap identifier. *)

  NEWHEAP(FIRST_HEAP);              (* Allocates a heap.          *)

  NEW(PHRASE_A);                    (* Allocates a phrase in      *)
                                   (* the default heap.          *)

  PHRASE_A@ := 'Now is the time ';

  MARK(MARK_ONE);                   (* Creates a mark in the      *)
                                   (* default heap.              *)

  NEW(PHRASE_B);                    (* Allocates another phrase   *)
                                   (* in the default heap.       *)

  PHRASE_B@ := 'for all good men ';

  USEHEAP(FIRST_HEAP);              (* Makes FIRST_HEAP the     *)
                                   (* current heap.              *)

  NEW(PHRASE_C);                    (* Allocates a phrase in      *)
                                   (* FIRST_HEAP.                *)

  PHRASE_C@ := 'to come to the aid ';

```

Figure 203 (Part 1 of 2). Example of the NEWHEAP Procedure

```

NEW(PHRASE_D);                (* Allocates a phrase in *)
                              (* FIRST_HEAP. *)
PHRASE_D@ := 'of their party.';

RELEASE(MARK_ONE);            (* Frees everything in the *)
                              (* heap containing MARK_ONE *)
                              (* (the default heap) since *)
                              (* MARK_ONE was done; thus, *)
                              (* this frees PHRASE_B. *)

DISPOSE(PHRASE_D);            (* Frees PHRASE_D from *)
                              (* FIRST_HEAP. *)
DISPOSEHEAP(FIRST_HEAP);      (* Frees everything in *)
                              (* FIRST_HEAP; note that *)
                              (* disposing of the entire *)
                              (* heap made the DISPOSE on *)
                              (* the previous line *)
                              (* unnecessary. *)

END.

```

Figure 203 (Part 2 of 2). Example of the NEWHEAP Procedure

Notes to Figure 203:

- The NEWHEAP creating FIRST_HEAP did not make FIRST_HEAP the current heap. To allocate dynamic variables from a heap created by a call to NEWHEAP, that heap must be made the current heap by calling USEHEAP.
- At the end of the program, PHRASE_A is the only dynamic variable not deallocated.

ODD Function

ODD returns TRUE if an integer is odd, or FALSE if it is even. Figure 204 shows the definition of the ODD function.

```

FUNCTION ODD( i : INTEGER )
              : BOOLEAN;

```

Where Represents
i An integer expression

Figure 204. Definition of the ODD Function

ORD Function

ORD returns an integer that corresponds to an ordinal value. ORD also works with pointers.

Figure 205 shows the definition of the ORD function.

```
FUNCTION ORD( s : ordinal-type )
           : INTEGER;
```

Where Represents

s An ordinal type or pointer expression

Figure 205. Definition of the ORD Function

If the operand is of type CHAR, the value returned is the position in the EBCDIC character set for the character operand. If the operand is an enumerated scalar, ORD returns the position in the enumeration (beginning at zero). For example, if COLOR = (RED, YELLOW, BLUE), then ORD(RED) is 0 and ORD(BLUE) is 2. If the operand is a pointer, the function returns the machine address of the dynamic variable referenced by the pointer.

Note: Although pointers can be converted to integers, there is no function provided to convert an integer to a pointer.

PACK Procedure

PACK copies elements from the unpacked source array to the packed target array, starting with the specified element of the source array. The types of the elements of the two arrays must be identical. However, in LANGLVL(EXTENDED), if the array elements are subranges, they need only have identical bounds. Figure 206 shows the definition of the PACK procedure.

```
PROCEDURE PACK( CONST source : array-type;
                index  : index-of-source;
                VAR  target : pack-array-type);
```

Where Represents

source An array

index An expression that is compatible with the index of *source*

target A packed array variable

Figure 206. Definition of the PACK Procedure

It is an error if the number of elements in the packed target array is greater than the number of elements used in the source array. See Figure 207 on page 151.

Assuming:

```
A : ARRAY[M..N] OF T;
Z : PACKED ARRAY[U..V] OF T;
```

Then:

```
PACK(A, I, Z);
```

is equivalent to:

```
K := I;
FOR J := LBOUND(Z) TO HBOUND(Z) DO
BEGIN
  Z[J] := A[K];
  IF J <> HBOUND(Z) THEN
    K:= SUCC(K);
END;
```

where J and K are temporary variables.

Figure 207. Example of the PACK Procedure

PAGE Procedure

PAGE causes a skip to the top of the next page when a text file is printed.

Figure 208 shows the definition of the PAGE procedure.

```
PROCEDURE PAGE( VAR f : TEXT );
```

Where Represents

f Optional variable of type TEXT; the default is the standard file variable OUTPUT

Figure 208. Definition of the PAGE Procedure

PARMS Function

PARMS returns a string that was associated with the invocation of the VS Pascal main program. Figure 209 shows the definition of the PARMS function.

```
FUNCTION PARMS : STRING;
```

Figure 209. Definition of the PARMS Function

PDSIN Procedure

PDSIN opens a member in a library (partitioned) file for input. This procedure is semantically equivalent to doing a RESET on a member of a specified library.

Figure 210 on page 152 shows the definition of the PDSIN procedure.

```
PROCEDURE PDSIN( VAR f   : filetype;  
                CONST s : STRING);
```

Where	Represents
-------	------------

<i>f</i>	A file variable
----------	-----------------

<i>s</i>	An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for information about these options)
----------	---

Figure 210. Definition of the PDSIN Procedure

PDSOUT Procedure

PDSOUT opens a member in a library (partitioned) file for output. This procedure is semantically equivalent to doing a REWRITE on a member of a library.

Figure 211 shows the definition of the PDSOUT procedure.

```
PROCEDURE PDSOUT( VAR f   : filetype;  
                 CONST s : STRING);
```

Where	Represents
-------	------------

<i>f</i>	A file variable
----------	-----------------

<i>s</i>	An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for information about these options)
----------	---

Figure 211. Definition of the PDSOUT Procedure

PRED Function

PRED returns the predecessor value of an ordinal expression. Figure 212 shows the definition of the PRED function.

```
FUNCTION PRED(s : ordinal-type )  
          : ordinal-type;
```

Where	Represents
-------	------------

<i>s</i>	An ordinal expression
----------	-----------------------

Figure 212. Definition of the PRED Function

The PRED of the first element of an ordinal type is an error. The PRED of an integer is equivalent to subtracting one from the value of the integer.

Figure 213 shows an example of the PRED function.

```

TYPE
  NEPHEWS = (HUEY, DUEY, LOUIE);

PRED(DUEY)      (* yields HUEY *)
PRED(HUEY)      (* is an error *)
PRED(TRUE)      (* yields FALSE *)
PRED('B')       (* yields 'A' *)
PRED(I)         (* yields I-1 *)
PRED(3.0)       (* is an error *)

```

Figure 213. Example of the PRED Function

PUT Procedure

PUT places the contents of the file pointer into the current position and positions the file to its next element. The file must have been previously opened for output. Figure 214 shows the definition of the PUT procedure.

```

PROCEDURE PUT( VAR f : filetype );

```

Where	Represents
<i>f</i>	A file variable

Figure 214. Definition of the PUT Procedure

Restriction: PUT cannot write DBCS data to a text file.

QUERYHEAP Procedure

QUERYHEAP returns the heap-id of the current heap. Figure 215 shows the definition of the QUERYHEAP procedure.

```

PROCEDURE QUERYHEAP( VAR p : pointer);

```

Where	Represents
<i>p</i>	Any pointer

Figure 215. Definition of the QUERYHEAP Procedure

After invocation, *p* contains the current heap-id. If an active heap does not exist, QUERYHEAP returns NIL.

See Figure 203 on page 148 to see how QUERYHEAP is used in conjunction with NEWHEAP, USEHEAP, and DISPOSEHEAP.

RANDOM Function

RANDOM returns a pseudo-random number in the range > 0.0 and < 1.0 . Figure 216 shows the definition of the RANDOM function.

```
FUNCTION RANDOM( s : INTEGER ) : REAL;
```

Where	Represents
s	An integer seed expression

Figure 216. Definition of the RANDOM Function

If you pass a seed value of 0, RANDOM generates the next number from the previous seed. Thus, the general way to use this function is to pass it a non-zero seed on the first invocation and a zero value thereafter. RANDOM always returns the same value when passed a non-zero seed.

READ Procedure (for Record Files)

READ reads data from record files. Each call to READ reads one record from the specified file into each variable in the call. Figure 217 shows the definition of the READ procedure (for record files).

```
PROCEDURE READ( VAR f      : FILE OF t;
                v1,v2...: t;
```

Where	Represents
f	A record file variable
v1,v2...	List of variables whose type matches the file component type of f

Figure 217. Definition of the READ Procedure (for Record Files)

Figure 218 shows an example of the READ procedure for record files.

Coding:

```
READ(f,v)
```

is equivalent to this compound statement:

```
BEGIN
  v := f@;
  GET(f);
END;
```

Figure 218. Example of the READ Procedure for Record Files

You can use more than one variable on each call by separating each variable with a comma. The effect is the same as multiple calls to READ. This implies that the variables are read in a left-to-right order. Figure 219 on page 155 shows an example of multiple variables on READ.

Coding:

```
READ(f,v1,v2);
```

is equivalent to:

```
BEGIN
  READ(f,v1);
  READ(f,v2);
END;
```

Figure 219. Example of Multiple Variables on READ

Data in the file is expected to be in its internal representation.

READ and READLN Procedures (for Text Files)

READ and READLN read data from a text file.

READ reads character data from a text file and converts the character data to conform to the type of the operand(s). The file parameter is optional; the default file is INPUT.

READLN reads data (if any variables are specified) the same way as READ, and then moves the file pointer to the beginning of the next line.

Figure 220 shows the definition of the READ and READLN procedures (for text files).

```
PROCEDURE READ( VAR f : TEXT;
                VAR v : see below);
```

```
PROCEDURE READLN( VAR f : TEXT;
                  VAR v : see below);
```

```
PROCEDURE READLN( VAR f : TEXT);
```

Where Represents

f An optional text file; the default is the predefined file INPUT

v List of variables (optional for READLN):

- CHAR (or subrange)
- DBCS fixed string
- GCHAR
- GSTRING
- INTEGER (or subrange)
- REAL
- SBCS fixed string
- SHORTREAL
- STRING

Figure 220. Definition of the READ and READLN Procedures (for Text Files)

Figure 221 shows an example of the READLN procedure.

Assume the data is:

36 24 ABCDEFGHIJKLMNOPQRSTUVWXYZ

With the following example:

```
VAR
  I,J: INTEGER;
  S: STRING(100);
  CH: CHAR;
  CC: PACKED ARRAY[1..10] OF CHAR;
  F: TEXT;
  .
  .
BEGIN
  READLN(F,I,J,CH,CC,S);
END;
```

The variables would be assigned:

I	36
J	24
CH	' '
CC	'ABCDEFGHIJ'
S	'KLMNOPQRSTUVWXYZ'
LENGTH(S)	16

Figure 221. Example of the READLN Procedure

You can use more than one variable on each call by separating each variable with a comma. The effect is the same as multiple calls to READ. This implies that the variables are read in a left-to-right order.

Figure 222 on page 157 shows an example of multiple variables on READ and READLN.

Coding:

```
READLN(f,v1,v2,v3);
```

is equivalent to:

```
BEGIN
  READ(f,v1);
  READ(f,v2);
  READ(f,v3);
  READLN(f);
END;
```

Coding:

```
READ(f,v1,v2);
```

is equivalent to:

```
BEGIN
  READ(f,v1);
  READ(f,v2);
END;
```

Figure 222. Example of Multiple Variables on READ and READLN

Reading Variables with a Length: You can qualify a READ variable with a field length expression; for example,

```
READ(f,v:n)
```

where *v* is the variable being read and *n* is the field length expression. This expression specifies the number of characters in the input line to be processed for the variable *v*.

If during a read operation the field ends before the field length is completed, the reading operation stops. The next read operation begins at the first character following the field.

If a read operation ends before processing all characters of the field, it skips the rest of the field.

If the specified field length is negative, the absolute value of the length is used. If the specified field length is zero, the read is done as if no field length were used.

Figure 223 on page 158 shows an example of the READLN procedure with lengths.

Assume the data is:

```
36 24 ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

With the following example:

```
VAR
  I,J: INTEGER;
  S: STRING(100);
  CH: CHAR;
  CC: PACKED ARRAY[1..10] OF CHAR;
  F: TEXT;
.
.
BEGIN
  READLN(F,I:4,J:10,CH:J,CC,S);
END;
```

The variables would be assigned:

I	36
J	4
CH	'I'
CC	'MNOPQRSTU'
S	'WXYZ'
LENGTH(S)	4

Figure 223. Example of the READLN Procedure with Lengths

Reading CHAR Data: A variable of type CHAR is assigned the next character in the file.

Reading DBCS Fixed String Data: If the variable is declared as a PACKED ARRAY[1..N] OF GCHAR, or DBCS fixed string, a sequence of DBCS characters is read. To begin reading, the file pointer must point to a byte that is a shift-out character, or to two bytes that are a DBCS character. If the field width is smaller than the length of the string, or if the string is not filled, it will be padded with blanks.

Reading GCHAR Data: When reading GCHAR data, one DBCS character is read. To begin reading, the file pointer must point to a byte that is a shift-out character or to two bytes that are a DBCS character. You will get an error message if any SBCS character, excluding shift-in and shift-out pairs, are found before the field width is exhausted.

Reading GSTRING Data: When reading GSTRING data, a sequence of DBCS characters is read. To begin reading, the file pointer must point to a byte that is a shift-out character or to two bytes that are a DBCS character. Reading stops when:

- The variable is filled.
- A shift-in character is followed by the end-of-line condition; it is an error if the shift-in character is missing.
- A shift-in character is followed by an SBCS character; it is an error if the shift-in character is missing.
- The field width is exhausted; it is an error if any SBCS characters are found, excluding shift-out and shift-in characters, before the field width is exhausted.

Figure 224 on page 159 shows an example of reading GCHAR and GSTRING data.

```

VAR
  F : TEXT;
  C : GCHAR;
  G1, G2 : GSTRING(3);

BEGIN
  RESET(F);
  READLN(F,C,G1,G2:2);
END;

```

If the input file contains:

<.A.B.C.D.E.F.G>

then:

```

C = '<.A>'G      (* yields .A stored in C *)
G1 = '<.B.C.D>'  (* yields .B.C.D stored in G1 *)
G2 = '<.E.F>'    (* yields .E.F stored in G2 *)

```

If the input file contains:

<.A><.B><.C>xxx

then:

```

C = '<.A>'G      (* yields .A stored in C *)
G1 = '<.B.C>'G    (* yields .B.C stored in G *)

```

Error reading G2 because the file had no DBCS characters.

If the input file contains:

<.A><.B.C.D.E>xxx

then:

```

C = '<.A>'G      (* yields .A stored in C *)
G1 = '<.B.C.D>'G  (* yields .B.C.D stored in G *)

```

Error reading G2 because DBCS characters ended before the field length ended.

Figure 224. Example of Reading GCHAR and GSTRING Data

Reading Integer Data: Integer data is read by skipping leading blanks and end-of-lines, reading an optional sign followed by one or more numeric characters until a non-numeric character is found. If the characters read do not form a valid integer, a run-time error will occur.

Reading Real and Shortreal Data: Real and shortreal data is read by skipping leading blanks and end-of-lines. VS Pascal reads characters until it finds one that cannot be in a real or shortreal number. If the characters read do not form a valid real or shortreal, a run-time error will occur.

Reading SBCS Fixed String Data: If the variable is declared as a PACKED ARRAY[1..N] OF CHAR, or SBCS fixed string, characters are stored into each element of the array. This is equivalent to a loop ranging from the lower bound of the array to the upper bound, performing a read operation for each element. If the

READSTR Procedure

end-of-line condition becomes true before the variable is filled, the rest of the variable is filled with blanks.

Reading String and Mixed String Data: Characters are read into a string variable until the variable has reached its maximum length or until the end of the line is reached.

When reading mixed string data, a specified field width applies only to the number of SBCS characters to be read from the file. READ does not check for valid DBCS characters.

Figure 225 shows an example of reading mixed string data.

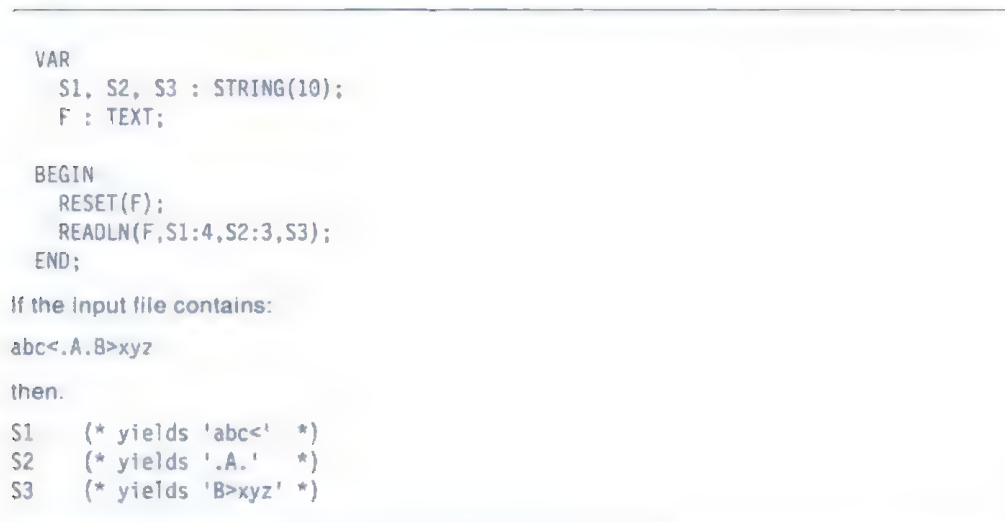


Figure 225. Example of Reading Mixed String Data

For more information on READ and READLN, see *VS Pascal Application Programming Guide*.

READSTR Procedure

READSTR reads character data from a source string into one or more variables.

The actions of READSTR are identical to those of READ except that the source data is extracted from a string expression instead of a text file. See "READ and READLN Procedures (for Text Files)" on page 155. READSTR is especially useful for converting a string to a different type.

Figure 226 on page 161 shows the definition of the READSTR procedure.

```
PROCEDURE READSTR( CONST s : STRING;
                   VAR  v : see below);
```

Where Represents

s A string expression that is to be used for input

v A list of one or more of these variable types:

- CHAR (or subrange)
- DBCS Fixed String
- GCHAR
- GSTRING
- INTEGER (or subrange)
- REAL
- SBCS Fixed String
- SHORTREAL
- STRING

Figure 226. Definition of the READSTR Procedure

As in the READ procedure, variables can be qualified with a field length expression (see Figure 227).

```
VAR
  I,J: INTEGER;
  S  : STRING(100);
  S1 : STRING(100);
  CH : CHAR;
  CC : PACKED ARRAY [1..10] OF CHAR;
.
BEGIN
  S := '36 245ABCDEFGHJK';
  READSTR(S,I,J:3,CH,CC:5,S1);
END;
```

The variables would be assigned:

I	36
J	24
CH	'5'
CC	'ABCDE'
S1	'FGHIJK';
LENGTH(S1)	6

Figure 227. Example of the READSTR Procedure

Figure 228 on page 162 shows an example of coding that has the same effect as READSTR

Coding:

```
READSTR(s,v1,v2);
```

has the same effect as coding:

```
WRITE(f,s);
READ(f,v1,v2);
```

where *f* is an optional text file.

Figure 228. Example of Code Equivalents to READSTR

Note: An error results if all the string characters are read before all the variables are filled.

RELEASE Procedure

RELEASE deallocates groups of dynamic variables delimited by the MARK routine. When a pointer returned from MARK is passed to RELEASE, all dynamic variables allocated in the heap containing the mark, and allocated since that call to MARK, are deallocated (as though a call to the DISPOSE routine were made for each variable). RELEASE then sets the pointer variable to NIL.

Figure 229 shows the definition of the RELEASE procedure.

```
PROCEDURE RELEASE( VAR p : pointer );
```

Where Represents

p A pointer returned from a call to MARK

Figure 229. Definition of the RELEASE Procedure

In this simple example,

```
MARK(X)
NEW(A)
NEW(B)
MARK(Y)
NEW(C)
```

the call RELEASE(Y) would deallocate only the dynamic variable pointed to by C. The call RELEASE(X) would free the dynamic variables pointed to by A, B, and C.

After you free a subheap with RELEASE, all of the pointers that once referenced dynamic variables in that subheap are undefined. If you try to reference these pointers later, VS Pascal can act unpredictably.

Figure 177 on page 137 shows how to use MARK and RELEASE within a single heap. See Figure 203 on page 148 to see how RELEASE is used with multiple heaps.

RESET Procedure

RESET opens a file for input. RESET positions the file pointer to the beginning of the file and prepares the file to be used for input. After you invoke RESET, the file pointer is pointing to the first data element of the file. This procedure can be thought of as:

1. Closing the file (if open)
2. Rewinding the file
3. Opening the file for input
4. Getting the first component of the file.

Figure 230 shows the definition of the RESET procedure.

```
PROCEDURE RESET( VAR f : filetype;
                  CONST s : STRING);
```

Where	Represents
<i>f</i>	A file variable
<i>s</i>	An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for further information about the options for opening files)

Figure 230. Definition of the RESET Procedure

RETCODE Procedure

RETCODE returns a return code to the caller of the VS Pascal program. The value of the operand will be returned to the system when an exit is made from the main program. If this routine is called several times, only the last value specified will be passed back to the system.

Figure 231 shows the definition of the RETCODE procedure.

```
PROCEDURE RETCODE( retvalue : INTEGER );
```

Where	Represents
<i>retvalue</i>	The return code to be passed to the caller of the VS Pascal program; the value is system dependent

Figure 231. Definition of the RETCODE Procedure

The VS Pascal run-time environment will set the return code when a run-time error occurs. The return code passed back will be the maximum of either the error return code or any previous setting.

Restriction: Passing a negative value to RETCODE can have unpredictable results.

REWRITE Procedure

REWRITE opens a file for output and erases the previous contents of the file. The procedure positions the file pointer to the beginning of the file and prepares the file to be used for output. This procedure can be thought of as:

1. Closing the file (if open)
2. Rewinding the file
3. Opening the file for output.

Figure 232 shows the definition of the REWRITE procedure.

```

PROCEDURE REWRITE( VAR f   : filetype;
                   CONST s : STRING);

Where  Represents
f       A file variable
s       An optional string of file dependent options to be used in opening the file (see
        Appendix C, "Options for Opening Files" on page 252 for further information
        about the options for opening files)

```

Figure 232. Definition of the REWRITE Procedure

RINDEX Function

RINDEX returns the position of the last occurrence of the second string within the first string. If the second string does not exist in the first string, RINDEX returns a zero. If the second string is null, LENGTH(s) + 1 will always be returned. Figure 233 shows the definition of the RINDEX function.

```

FUNCTION RINDEX( CONST source : STRING;
                 CONST lookup  : STRING )
                : 0..32768;

FUNCTION RINDEX( CONST source : GSTRING;
                 CONST lookup  : GSTRING )
                : 0..16383;

Where  Represents
source  A string expression to which lookup is compared
lookup  A string expression which will be compared to source

```

Figure 233. Definition of the RINDEX Function

Note: Although RINDEX is better suited to pure SBCS and pure DBCS strings, it can be used for mixed strings. RINDEX treats mixed strings in a byte-oriented manner. However, MRINDEX is usually used for mixed strings.

Figure 234 shows examples of the RINDEX function.

```

S = 'ABCABCABC'
RINDEX(S, 'BC')      (* yields 8 *)
S = 'ABCABCABC'
RINDEX(S, 'X')       (* yields 0 *)

S = '<.A.B.C.A.B.C>ABC'
RINDEX(S, '<.B.C>')    (* yields 5 *)
S = '<.A.B.C.A.B.C>ABC'
RINDEX(S, 'C')       (* yields 9 *)

S = '<.A.B.A>'G
RINDEX(S, '<.A>'G)     (* yields 3 *)

```

Figure 234. Examples of the RINDEX Function

ROUND Function

ROUND converts a real expression to an integer expression by rounding the operand. This function is equivalent to:

```
IF R > 0.0 THEN
  ROUND := TRUNC(R + 0.5)
ELSE
  ROUND := TRUNC(R - 0.5);
```

Figure 235 shows the definition of the ROUND function.

```
FUNCTION ROUND( r : REAL )
  : INTEGER;

FUNCTION ROUND( s : SHORTREAL )
  : INTEGER;
```

Where **Represents**

r An expression with a real value

s An expression with a shortreal value

Figure 235. Definition of the ROUND Function

Figure 236 shows an example of the ROUND function.

```
ROUND( 1.0)   (* IS 1 *)
ROUND( 1.1)   (* IS 1 *)
ROUND( 1.5)   (* IS 2 *)
ROUND( 1.9)   (* IS 2 *)
ROUND( 0.0)   (* IS 0 *)
ROUND(-1.0)   (* IS -1 *)
ROUND(-1.1)   (* IS -1 *)
ROUND(-1.5)   (* IS -2 *)
ROUND(-1.9)   (* IS -2 *)
```

Figure 236. Example of the ROUND Function

RPAD Procedure

RPAD pads or truncates a string on the right. RPAD manipulates pure SBCS and pure DBCS strings.

Figure 237 on page 166 shows the definition of the RPAD procedure.

```

PROCEDURE RPAD( VAR s : STRING;
                l : INTEGER;
                c : CHAR);

PROCEDURE RPAD( VAR s : GSTRING;
                l : INTEGER;
                c : GCHAR);

```

Where	Represents
s	The string to be padded
l	The final length of s
c	The pad character

Figure 237. Definition of the RPAD Procedure

Note to Figure 237: If LENGTH(s) is greater than l, the characters are truncated on the right. If LENGTH(s) is less than l, s is extended with the character c on the right.

Figure 238 shows an example of the RPAD procedure.

```

S := 'ABCDEF';
RPAD(S, 10, '$');    (* yields 'ABCDEF$$$$' in S *)
S := 'ABCDEF';
RPAD(S, 5, '$');     (* yields 'ABCDE' in S *)

G := '<.A.B.C.D.E.F>'G;
RPAD(G, 10, '<.$>'G) (* yields '<.A.B.C.D.E.F.$.$.$>'G in G *)
G := '<.A.B.C.D.E.F>'G;
RPAD(G, 5, '<.$>'G)  (* yields '<.A.B.C.D.E>'G in G *)

```

Figure 238. Example of the RPAD Procedure

Note: Because RPAD is now a predefined routine, %INCLUDE STRING is no longer required to invoke RPAD.

SEEK Procedure

SEEK positions a file pointer to a specified element after the file has been opened by the RESET, REWRITE, or UPDATE routines. SEEK specifies the number of the next file component that GET or PUT will operate on. File components have an origin of 1.

Figure 239 shows the definition of the SEEK procedure.

```

PROCEDURE SEEK( VAR f : filetype;
                n : INTEGER);

```

Where	Represents
f	A record file variable
n	Component number of the file

Figure 239. Definition of the SEEK Procedure

Restriction: SEEK can be used only with record files.

Note: The file buffer value is undefined after calling SEEK.

SIN Function

SIN computes the sine of a floating-point number representing an angle in radians.

Figure 240 shows the definition of the SIN function.

```
FUNCTION SIN( x : REAL )
           : REAL;
```

Where	Represents
--------------	-------------------

x	An expression that evaluates to a real value
---	--

Figure 240. Definition of the SIN Function

Real functions will accept integer and shortreal arguments. See "Type Compatibility" on page 46 for more information.

SIZEOF Function

SIZEOF returns the amount of storage (in bytes) needed to contain a variable of the type specified. Figure 241 shows the definition of the SIZEOF function.

```
FUNCTION SIZEOF( s : any-type)
           : INTEGER;
```

```
FUNCTION SIZEOF( s : record-type;
                t1,t2,... : ordinal-type)
           : INTEGER;
```

Where	Represents
--------------	-------------------

s	A type or variable identifier
---	-------------------------------

t1, t2	Ordinal constants representing tag fields
--------	---

Figure 241. Definition of the SIZEOF Function

If the parameter *s* refers to a record that has a variant part, and if no tag values are specified, then the storage required for the record with the largest variant will be returned.

If *s* is a record variable or a type identifier of a record, it can be followed by a tag list that defines a particular variant configuration of the record. In this case, the function will return the amount of storage required to contain the record with the specified variants active.

SQR Function

SQR computes the square of the argument. The function returns the same type as the argument. Figure 242 on page 168 shows the definition of the SQR function.

```

FUNCTION SQR( i : INTEGER )
            : INTEGER;

FUNCTION SQR( r : REAL )
            : REAL;

FUNCTION SQR( s : SHORTREAL )
            : SHORTREAL;

```

Where	Represents
<i>i</i>	An integer expression
<i>r</i>	A real expression
<i>s</i>	A shortreal expression

Figure 242. Definition of the SQR Function

SQRT Function

SQRT computes the square root of a number. If the argument is less than zero, a run-time error is produced.

Figure 243 shows the definition of the SQRT function.

```

FUNCTION SQRT( r : REAL )
            : REAL;

Where Represents
r         A real expression

```

Figure 243. Definition of the SQRT Function

Real functions will accept integer and shortreal arguments. See "Type Compatibility" on page 46 for more information.

STOGSTR Function

STOGSTR converts a STRING to a GSTRING. STOGSTR removes one shift-out from the beginning of the STRING, and one shift-in from the end of the STRING. If the STRING contains any SBCS characters, VS Pascal raises an error.

Figure 244 shows the definition of the STR function.

```

FUNCTION STOGSTR(s : STRING)
            : GSTRING;

Where Represents
s         A string containing only DBCS characters and shifts

```

Figure 244. Definition of the STR Function

Figure 245 on page 169 shows an example of the STOGSTR function.

```

VAR
  G : GSTRING(4);
  S : STRING(10);

BEGIN
  S := '<.A.B>'      (* <.A.B> is stored in S      *)
  G := STOGSTR(S);  (* .A.B is stored in G      *)

  S := '<.A>bc'      (* <.A>bc is stored in S      *)
  G := STOGSTR(S);  (* Error (S contains SBCS characters) *)
END;

```

Figure 245. Example of the STOGSTR Function

STR Function

STR converts a CHAR or an SBCS fixed string to a STRING. STR can also be applied to a STRING, but no operation is performed. Figure 246 shows the definition of the STR function.

```

FUNCTION STR( x : CHAR)
  : STRING;

FUNCTION STR( x : PACKED ARRAY[1..n] OF CHAR)
  : STRING;

FUNCTION STR( x : STRING)
  : STRING;

```

Where Represents

x An SBCS character, SBCS fixed string, or SBCS variable-length string

Figure 246. Definition of the STR Function

Figure 247 shows an example of the STR function.

```

VAR
  SC : CHAR;
  SA : PACKED ARRAY[1..4] of CHAR;
  S4 : STRING;

BEGIN
  SC := 'A';      (* 'A' is stored in SC *)
  S4 := STR(SC);  (* 'A' is stored in S4 *)
  SA := 'AB';     (* 'AB' is stored in SA *)
  S4 := STR(SA);  (* 'AB' is stored in S4 *)
END;

```

Figure 247. Example of the STR Function

SUBSTR Function

SUBSTR returns a specified portion of the source string.

Figure 248 shows the definition of the SUBSTR function.

```

FUNCTION SUBSTR( CONST source : STRING;
                  start  : INTEGER;
                  len    : INTEGER) : STRING;

FUNCTION SUBSTR( CONST source : GSTRING;
                  start  : INTEGER);
                  len    : INTEGER) : GSTRING;

```

Where Represents

source	A string expression from which a substring will be returned.
start	An integer expression that specifies the starting position within the source from which the substring is to be extracted. The first character of the source string is at position 1.
len	An optional integer expression that determines the length of the substring. If <i>len</i> is omitted, it defaults to <code>LENGTH(s) - start + 1</code> ; in other words, the substring returned will be the remaining portion of the source string from position <i>start</i> .

Figure 248. Definition of the SUBSTR Function

Note: Although SUBSTR is better suited to pure SBCS and pure DBCS strings, it can be used for mixed strings. SUBSTR manipulates mixed strings in a byte-oriented manner. However, MSUBSTR is usually used for mixed strings.

Usage: To avoid an error message at run time:

- *start* must be greater than 0.
- *len* must be greater than or equal to 0. If *len* is 0, a null string is returned.
- *start* + *len* - 1 must be less than or equal to the current length of the string.

Figure 249 shows an example of the SUBSTR function.

```

SUBSTR('ABCDE',2,3)      (* yields 'BCD'    *)
SUBSTR('ABCDE',1,3)      (* yields 'ABC'    *)
SUBSTR('ABCDE',4)        (* yields 'DE'     *)
SUBSTR('ABCDE',1)        (* yields 'ABCDE'  *)
SUBSTR('ABCDE',6,0)      (* returns ''      *)
SUBSTR('ABCDE',2,5)      (* is an error     *)

SUBSTR('<.A.B>CDE',1,6)    (* yields '<.A.B>'  *)

SUBSTR('<.A.B.C>'G,2,2)    (* yields '<.B.C>'G *)

```

Figure 249. Example of the SUBSTR Function

SUCC Function

SUCC returns the successor value of an ordinal expression. Figure 250 shows the definition of the SUCC function.

```
FUNCTION  SUCC(s : ordinal-type )
           : ordinal-type;
```

Where	Represents
s	An ordinal expression

Figure 250. Definition of the SUCC Function

The SUCC of the last element of an enumerated scalar is an error. The SUCC of an integer is equivalent to adding one to the value of the integer.

Figure 251 shows an example of the SUCC function.

```
TYPE
  NEPHEWS = (HUEY, DUEY, LOUIE);

SUCC(DUEY)      (* yields LOUIE *)
SUCC(LOUIE)     (* is an error *)
SUCC(FALSE)     (* yields TRUE *)
SUCC('B')       (* yields 'C' *)
SUCC(I)         (* yields I+1 *)
SUCC(3.0)       (* is an error *)
```

Figure 251. Example of the SUCC Function

TERMIN Procedure

TERMIN opens the designated file for input from your terminal. Figure 252 shows the definition of the TERMIN procedure.

```
PROCEDURE  TERMIN( VAR f : TEXT;
                  CONST s : STRING);
```

Where	Represents
f	A text file variable
s	An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for further information about the options for opening files)

Figure 252. Definition of the TERMIN Procedure

TERMOUT Procedure

TERMOUT opens the designated file for output to your terminal. Figure 253 on page 172 shows the definition of the TERMOUT procedure.

```
PROCEDURE TERMOUT( VAR f : TEXT;
                  CONST s : STRING);
```

Where Represents

f A text file variable

s An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for further information about the options for opening files)

Figure 253. Definition of the TERMOUT Procedure

TOKEN Procedure

Starting from a given position, TOKEN scans a string for a token, and returns the token in an ALPHA array. Figure 254 shows the definition of the TOKEN procedure.

```
PROCEDURE TOKEN( VAR pos : INTEGER;
                CONST source : STRING;
                VAR result : ALPHA );
```

Where Represents

pos An integer corresponding to the position in the source string where the search for the token begins. The value of this integer gets updated to reflect the starting position for subsequent calls to TOKEN.

source A string expression that contains the data from which a token is to be extracted.

result The resulting token.

Figure 254. Definition of the TOKEN Procedure

When TOKEN scans a string, it ignores leading, multiple, and trailing blanks. If there is no token in the string, the value of the first parameter *pos* is set to $\text{LENGTH}(s) + 1$, and the *result* parameter is set to one blank.

If the token is longer than ALPHALEN, only the first ALPHALEN characters are returned, but *pos* is updated to point past the entire token and any trailing blanks.

A token can be any of the following:

- VS Pascal identifier, any number of alphanumeric characters, "\$", or an underscore. The first letter must be alphabetic or a "\$".
- VS Pascal unsigned integer. (See "Types of Constants" on page 36.)
- The following special symbols:

+	-	*	/	->	@	¢
=	<>	<	<=	>=	>	!
{	}	[]	'	"	;
	&	&&		¬	¬=	#
:	;	:=	.	,	..	><
{	}	(*	*)	/*	*/	
(.	.)	<<	>>			

- Any other single character not listed here.

Figure 255 shows an example of the TOKEN procedure.

```

I := 2;
TOKEN(I, ' ', Token+, RESULT) (* I is set to 8 *)
                                (* RESULT is set to 'Token' *)

```

Figure 255. Example of the TOKEN Procedure

Note to Figure 255: TOKEN would return the same value if I were set to 3; that is, the leading blanks are ignored.

TRACE Procedure

TRACE writes the current list of procedures and functions pending execution (the save chain). Each line of the listing contains:

- The name of the routine
- The statement number where the call took place
- The return address in hexadecimal
- The name of the unit that contained the calling procedure.

Figure 256 shows the definition of the TRACE procedure.

```
PROCEDURE TRACE( VAR f : TEXT);
```

Where Represents

f A text file that will receive the trace listing

Figure 256. Definition of the TRACE Procedure

TRIM Function

TRIM returns the value of the specified parameter with all trailing blanks removed.

Note: Although TRIM is better suited to pure SBCS and pure DBCS strings, it can be used for mixed strings. TRIM manipulates mixed strings in a byte-oriented manner. However, MTRIM is usually used for mixed strings.

Figure 257 shows the definition of the TRIM function.

```

FUNCTION TRIM( CONST source : STRING )
    : STRING;

FUNCTION TRIM( CONST source : GSTRING )
    : GSTRING;

```

Where Represents
source A string expression

Figure 257. Definition of the TRIM Function

Figure 258 shows an example of the TRIM function.

```

TRIM(' A B ')      (* yields ' A B' *)
TRIM(' ')          (* yields '' *)

TRIM('<.A.b.b.b>'G)  (* yields '<.A>'G *)

```

Figure 258. Example of the TRIM Function

TRUNC Function

TRUNC converts a real expression to an integer expression by truncating the operand toward zero. Figure 259 shows the definition of the TRUNC function.

```

FUNCTION TRUNC( r : REAL )
    : INTEGER;

FUNCTION TRUNC( s : SHORTREAL )
    : INTEGER;

```

Where Represents
r An expression with a real value
s An expression with a shortreal value

Figure 259. Definition of the TRUNC Function

Figure 260 shows an example of the TRUNC function.

```

TRUNC( 1.0)  (* IS 1 *)
TRUNC( 1.1)  (* IS 1 *)
TRUNC( 1.5)  (* IS 1 *)
TRUNC( 1.9)  (* IS 1 *)
TRUNC( 0.0)  (* IS 0 *)
TRUNC(-1.0)  (* IS -1 *)
TRUNC(-1.1)  (* IS -1 *)
TRUNC(-1.5)  (* IS -1 *)
TRUNC(-1.9)  (* IS -1 *)

```

Figure 260. Example of the TRUNC Function

UNPACK Procedure

UNPACK copies elements from the packed source array to the unpacked target array, starting with the specified element of the target array. The types of the elements of the two arrays must be identical. However, in `LANGLVL(EXTENDED)`, if the array elements are subranges, they need only have identical bounds.

Figure 261 shows the definition of the UNPACK procedure.

```

PROCEDURE UNPACK( CONST source : pack-array-type;
                  VAR  target : array-type;
                  index  : index-of-target);

```

Where Represents

source A packed array

target An array variable

index An expression that is compatible with the index of *target*

Figure 261. Definition of the UNPACK Procedure

It is an error if the number of elements in the packed source array is greater than the number of elements used in the target array.

Figure 262 shows an example of the UNPACK procedure.

Assuming the following declarations:

```
A : ARRAY[M..N] OF T;
Z : PACKED ARRAY[U..V] OF T;
```

The example:

```
UNPACK(Z, A, I);
```

is equivalent to:

```
K := I;
FOR J := LBOUND(Z) TO HBOUND(Z) DO
  BEGIN
    A[K] := Z[J];
    IF J <> HBOUND(Z) THEN
      K := SUCC(K);
  END;
```

where J and K are temporary variables.

Figure 262. Example of the UNPACK Procedure

UPDATE Procedure

UPDATE opens a record file for both input and output (updating). A PUT operation replaces a file component obtained from a preceding GET operation. The execution of UPDATE causes an implicit GET of the first file component (as in RESET).

Figure 263 shows the definition of the UPDATE procedure.

```
PROCEDURE UPDATE( VAR f : filetype;
                  CONST s : STRING);
```

Where	Represents
--------------	-------------------

<i>f</i>	A record file variable.
----------	-------------------------

<i>s</i>	An optional string of file dependent options to be used in opening the file (see Appendix C, "Options for Opening Files" on page 252 for further information about the options for opening files)
----------	---

Figure 263. Definition of the UPDATE Procedure

Figure 264 shows an example of the UPDATE procedure.

```

VAR
  FILEVAR : FILE OF RECORD
          CNT : INTEGER;
          .
          .
          .
          END;

.
.
.
BEGIN
  UPDATE(FILEVAR); (*open and get *)
  WHILE NOT EOF(FILEVAR) DO
    BEGIN
      FILEVAR@.CNT := FILEVAR@.CNT+1;
      PUT(FILEVAR); (*update last element*)
      GET(FILEVAR); (*get next element*)
    END;
  END;

```

Figure 264. Example of the UPDATE Procedure

USEHEAP Procedure

USEHEAP sets the current heap using a previously set heap-id. Figure 265 shows the definition of the USEHEAP procedure.

```

PROCEDURE USEHEAP(p : pointer);

```

Where	Represents
<i>p</i>	A heap-id

Figure 265. Definition of the USEHEAP Procedure

Figure 203 on page 148 shows how USEHEAP is used in conjunction with NEWHEAP, QUERYHEAP, and DISPOSEHEAP.

WRITE Procedure (for Record Files)

WRITE writes data to record files. Each call to WRITE writes the value of each expression in the call to a new record in the specified file. Figure 266 shows the definition of the WRITE procedure (for record files).

```

PROCEDURE WRITE( VAR f : FILE OF t;
                 e1,e2....: t);

```

Where	Represents
<i>f</i>	A file variable
<i>e1, e2...</i>	A list of expressions whose types match the file component type of <i>f</i>

Figure 266. Definition of the WRITE Procedure (for Record Files)

Figure 267 shows an example of the WRITE procedure for record files.

Coding:

```
WRITE(f,e)
```

is equivalent to the compound statement:

```
BEGIN
  f@ := e;
  PUT(f);
END;
```

Figure 267. Example of the WRITE Procedure for Record Files

You can output more than one expression on each call by separating each expression with a comma. The effect is the same as multiple calls to WRITE. This implies that the variables are read in a left-to-right order. Figure 268 shows an example of multiple expressions on WRITE.

Coding:

```
WRITE(f,e1,e2);
```

is equivalent to:

```
BEGIN
  WRITE(f,e1);
  WRITE(f,e2);
END;
```

Figure 268. Example of Multiple Expressions on WRITE

Data is written to the file in its internal representation.

WRITE and WRITELN Procedures (for Text Files)

WRITE and WRITELN write data to a text file.

WRITE writes character data to a text file. The data is obtained by converting the expression to appropriate output form. The file parameter is optional; the default file is OUTPUT.

WRITELN writes out data (if any expressions are specified) the same way as WRITE, and then moves the file pointer to the beginning of the next line.

Figure 269 on page 179 shows the definition of the WRITE and WRITELN procedures (for text files).

```
PROCEDURE WRITE( VAR f : TEXT;
                 e : see below);
```

```
PROCEDURE WRITELN( VAR f : TEXT;
                   e : see below);
```

```
PROCEDURE WRITELN( VAR f : TEXT);
```

Where Represents

f An optional text file; the default is the predefined file OUTPUT

e List of expressions (optional for WRITELN):

BOOLEAN
 CHAR (or subrange)
 DBCS Fixed String
 GCHAR
 GSTRING
 INTEGER (or subrange)
 REAL
 SBOS Fixed String
 SHORTREAL
 STRING

Figure 269. Definition of the WRITE and WRITELN Procedures (for Text Files)

You can output more than one expression on each call by separating each expression with a comma. The effect is the same as multiple calls to WRITE. This implies that the variables are read in a left-to-right order.

Figure 270 shows an example of multiple expressions on WRITE and WRITELN.

Coding:

```
WRITELN(f,e1,e2,e3);
```

is equivalent to:

```
BEGIN
  WRITE(f,e1);
  WRITE(f,e2);
  WRITE(f,e3);
  WRITELN(f);
END;
```

Coding:

```
WRITE(f,e1,e2);
```

is equivalent to:

```
BEGIN
  WRITE(f,e1);
  WRITE(f,e2);
END;
```

Figure 270. Example of Multiple Expressions on WRITE and WRITELN

Writing Expressions with a Length: You can control the length of the resulting output to text files by specifying parameters on WRITE and WRITELN. Each expression in the procedure call can be represented in one of these forms:

- Form 1: *expr*
- Form 2: *expr* : *TotalWidth*
- Form 3: *expr* : *TotalWidth* : *FracDigits*

Expr represents the data to be placed in the file. The data is converted to character representations from its internal form.

TotalWidth and *FracDigits* must evaluate to an integer value. In Standard Pascal, *TotalWidth* and *FracDigits* must be greater than 0; in VS Pascal, any integer can be used.

The expression *TotalWidth* supplies the length of the field into which the data is written. If *TotalWidth* specifies a positive number, the data is placed in the field justified to the right edge of the field. If *TotalWidth* specifies a negative value, the data is justified to the left within a field whose length is $ABS(TotalWidth)$. If *TotalWidth* is 0:

- No characters are written for character and Boolean data
- All characters are written for integer data
- The format of floating-point data is unpredictable.

If *TotalWidth* is unspecified, as in Form 1, a default value is used. See Figure 271.

FracDigits, as in Form 3, can be specified only for expressions of type REAL or SHORTREAL. *FracDigits* controls the number of decimal places that will appear in the output.

Figure 271 shows the default field widths on WRITE and WRITELN.

Type of Expression	Default Value of TotalWidth
BOOLEAN	10
CHAR	1
DBCS fixed strings	Length of array
Fixed strings	Length of array
GCHAR	1
GSTRING	LENGTH(expression)
INTEGER	12
REAL	20 (scientific notation)
SHORTREAL	20 (scientific notation)
STRING	LENGTH(expression)

Figure 271. Default Field Widths on WRITE and WRITELN

Writing Boolean Data: Boolean data is written exactly the same as the character strings TRUE or FALSE would be (depending on the value of the expression). The data is placed in the field and justified according to the previously stated rules. If *TotalWidth* is zero, then no characters are written. Figure 272 on page 181 shows examples of writing Boolean data.

Call	Result
WRITE(TRUE:10)	'bbbbbbTRUE'
WRITE(TRUE:-10)	'TRUE'
WRITE(FALSE:2)	'FA'
WRITE(FALSE:0)	''

Figure 272. Examples of Writing Boolean Data

Writing CHAR Data: The value of *TotalWidth* is used to indicate the width of the field in which the character is to be placed. The character is placed in the field given by *TotalWidth*. If *TotalWidth* is zero, no characters are written. Figure 273 shows examples of writing CHAR data.

Call	Result
WRITE('A':6)	'bbbbbbA'
WRITE('A':-6)	'Abbbbbb'
WRITE('A':0)	''

Figure 273. Examples of Writing CHAR Data

Writing DBCS Fixed String Data: If $ABS(\textit{TotalWidth})$ is too small to hold the data, the string is truncated on the right. If *TotalWidth* is zero, no characters are written. When writing DBCS fixed string data, VS Pascal includes the shift-out and shift-in characters.

Figure 274 shows examples of writing DBCS fixed string data.

Call	Result
WRITE('<.A.B.C.D>'G:6)	'<.b.b.A.B.C.D>'
WRITE('<.A.B.C.D>'G:-6)	'<.A.B.C.D.b.b>'
WRITE('<.A.B.C.D>'G:2)	'<.A.B>'
WRITE('<.A.B.C.D>'G)	'<.A.B.C.D>'
WRITE('<.A.B.C.D>'G:0)	''

Figure 274. Examples of Writing DBCS Fixed String Data

Writing GCHAR Data: When writing GCHAR data, the field width is the number of characters written to the file. If the field width is zero, no characters are written.

VS Pascal creates a DBCS string, enclosed by a shift-out/shift-in pair, of the same length as specified on the field width.

Figure 275 on page 182 shows examples of writing GCHAR data.

Call	Result
WRITELN('<.A>'G:1)	'<.A>'
WRITELN('<.A>'G:-1)	'<.A>'
WRITELN('<.A>'G:0)	' '
WRITELN('<.A>'G:2)	'<.A.b>'
WRITELN('<.A>'G:-2)	'<.b.A>'

Figure 275. Examples of Writing GCHAR Data

Writing GSTRING Data: When writing GSTRING data, the field width is the number of characters written to the file. If the field width is zero, no characters are written.

Figure 276 shows examples of writing GSTRING data.

Call	Result
WRITELN('<.B.C.D>'G:1)	'<.B>'
WRITELN('<.B.C.D>'G:4)	'<.B.C.D.b>'
WRITELN('<.B.C.D>'G:0)	' '
WRITELN('<.B.C.D>'G:-4)	'<.b.B.C.D>'

Figure 276. Examples of Writing GSTRING Data

Writing Integer Data: The expression *Total/Width* represents the minimum width of the field in which the integer is to be placed. The value is converted to character format and placed in a field of the specified length. If the field is shorter than the size required to represent the value, the length of the field will be extended. If *Total/Width* is zero, all digits are written.

Figure 277 shows examples of writing integer data.

Call	Result
WRITE(1234:6)	'bb1234'
WRITE(1234:-6)	'1234bb'
WRITE(1234:1)	'1234'
WRITE(1234)	'bbbbbb1234'
WRITE(1234:-3)	'1234'
WRITE(1234:0)	'1234'

Figure 277. Examples of Writing Integer Data

Writing Real and Shortreal Data: Real and shortreal expressions can be written with any one of the three operand formats. If *Total/Width* is not specified (form 1), the result will be in scientific notation in a 20-character field.

If *Total/Width* is specified and *FracDigits* is not (form 2), the result will be in scientific notation, but the number of characters in the field will be the value of *Total/Width*. One decimal place is always generated, and the result is rounded to

the last displayed decimal place. The exception is when *TotalWidth* is zero, in which case the result is unpredictable.

If both *TotalWidth* and *FracDigits* are specified (form 3), the data will be written in fixed-point notation in a field with the length of *TotalWidth*. If *FracDigits* is positive, the specified number of digits to the right of the decimal point are written. If *FracDigits* equals zero, a decimal point is written, but no decimal places are written. If *FracDigits* is negative, the number is written using scientific notation as if *FracDigits* were not specified.

If *TotalWidth* is not large enough to fully represent the number, it will be extended appropriately. The real expression is always rounded to the last digit to be written.

Figure 278 shows examples of writing real data.

Call	Result
WRITE(3.14159)	'b3.1415900000000E+00'
WRITE(3.14159:10)	'b3.142E+00'
WRITE(3.14159:1)	'b3.1E+00'
WRITE(3.14159:0)	unpredictable results
WRITE(3.14159:10:4)	'bbbb3.1416'
WRITE(3.14159:-10:4)	'3.1416bbbb'
WRITE(3.14159:10:0)	'bbbbbbbbb3.'
WRITE(3.14159:10:-1)	'b3.142E+00'

Figure 278. Examples of Writing Real Data

Writing SBCS Fixed String Data: If $ABS(\textit{TotalWidth})$ is too small to hold the data, the string is truncated on the right. If *TotalWidth* is zero, no characters are written.

Figure 279 shows examples of writing SBCS fixed string data.

Call	Result
WRITE('ABCD':6)	'bbABCD'
WRITE('ABCD':-6)	'ABCDbb'
WRITE('ABCD':2)	'AB'
WRITE('ABCD':-2)	'AB'
WRITE('ABCD')	'ABCD'
WRITE('ABCD':0)	''

Figure 279. Examples of Writing SBCS Fixed String Data

Writing String and Mixed String Data: When writing string data, if $ABS(\textit{TotalWidth})$ is too small to hold the data, the string is truncated on the right. If *TotalWidth* is zero, then no characters are written.

Figure 280 shows examples of writing string data.

Call	Result
WRITE('ABCD':6)	'bbABCD'
WRITE('ABCD':-6)	'ABCDbb'
WRITE('ABCD':2)	'AB'
WRITE('ABCD':-2)	'AB'
WRITE('ABCD')	'ABCD'
WRITE('ABCD':0)	''

Figure 280. Examples of Writing String Data

When writing mixed string data, the field width is the number of SBCS characters written to the file. If the field width is zero, no characters are written.

Figure 281 shows examples of writing mixed string data.

Call	Result
WRITE('<.A.B>c':2)	'<.'
WRITE('<.A.B>c':0)	''

Figure 281. Examples of Writing Mixed String Data

For more information on the WRITE and WRITELN procedures (for text files), see the *VS Pascal Application Programming Guide*.

WRITESTR Procedure

WRITESTR converts expressions into character data and stores the data into a string variable.

The actions of WRITESTR are identical to those of WRITE, except that the target of the data is a string rather than a text file. (See "WRITE and WRITELN Procedures (for Text Files)" on page 178.) WRITESTR is especially useful for converting data into strings.

Figure 282 on page 185 shows the definition of the WRITESTR procedure.

```
PROCEDURE WRITESTR( VAR s : STRING;
                   e : see below);
```

Where	Represents
s	A string variable
e	A list of one or more of these expression types:
	BOOLEAN
	CHAR (or subrange)
	DBCS Fixed String
	GCHAR
	GSTRING
	INTEGER (or subrange)
	REAL
	SBCS Fixed String
	SHORTREAL
	STRING

Figure 282. Definition of the WRITESTR Procedure

If the string variable *s* appears in the expression list of WRITESTR, the value of *s* will be unpredictable.

As in the WRITE procedure, the expressions being converted can be qualified with a field length expression.

Figure 283 shows an example of the WRITESTR procedure.

```
VAR
  I,J: INTEGER;
  S : STRING(100);
  R : REAL;
  CH : CHAR;
  .
  .
BEGIN
  I := 10;
  J := -123;
  R := 3.14159;
  CH := '*';
  WRITESTR(S,I:3,J:5,'ABC',CH,
           R:5:2);
END;
```

The variable *S* is assigned:

```
' 10 -123ABC* 3.14'
```

Figure 283. Example of the WRITESTR Procedure

Figure 284 on page 186 shows an example of coding that has the same effect as WRITESTR.

Coding:

```
WRITESTR(s,e1,e2);
```

has the same effect as coding:

```
WRITELN(f,e1,e2);
READ(f,s);
```

where *f* is an optional text file.

Figure 284. Example of Code Equivalents to WRITESTR

Note: An error results if all variables are filled before all expressions are written.

Additional Routines

The routines in this chapter, with the exception of LPAD and RPAD, are not predefined and can be passed as parameters to other routines. You can access these routines by coding a %INCLUDE compiler directive in your source. The routines are:

CMS	ONERROR
ITOHs	PICTURE
LPAD	RPAD

CMS Procedure

CMS issues a CMS command.

Figure 285 shows the definition of the CMS procedure.

```
PROCEDURE CMS( CONST s : STRING;
                VAR rc : INTEGER);
EXTERNAL;
```

Where	Represents
<i>s</i>	A string that is to be executed
<i>rc</i>	The return code

Figure 285. Definition of the CMS Procedure

The string specified by *s* is passed to CMS (via SVC 202 or 204) to be executed. The command must not overlay the Pascal program (at X'20000' in VM/SP, for example), or issue a storage initialization (STRINIT). The command can run in the nucleus or transient area, or run as a nucleus extension. See the appropriate CMS manuals.

You must code the declaration as shown in Figure 286 on page 187, or use the %INCLUDE member named "CMS" provided in the VS Pascal library. This procedure can be used under CMS only.

Figure 286 on page 187 shows an example of the CMS procedure.

```
%INCLUDE CMS
.
CMS('CP Q T', RET);
```

Figure 286. Example of the CMS Procedure

Note: If you want to execute a CP command or an EXEC, the command string must begin with CP or EXEC, respectively.

ITOHS Function

ITOHS returns a string containing the hexadecimal representation of an integer.

Figure 287 shows the definition of the ITOHS procedure.

```
FUNCTION ITOHS( I : INTEGER)
              : STRING(8);
EXTERNAL;
```

Where	Represents
I	An integer expression

Figure 287. Definition of the ITOHS Function

You must code the declaration as shown in Figure 287, or use the %INCLUDE member named "CONVERT" provided in the VS Pascal library.

Figure 288 shows an example of the ITOHS function.

```
%INCLUDE CONVERT
.
.
WRITELN('The value ',I:0,
        ' is ', ITOHS(I),
        ' in hexadecimal.');
```

Figure 288. Example of the ITOHS Function

LPAD Procedure

LPAD pads or truncates a string on the left. It only works with string data; you must use the predefined LPAD procedure to manipulate DBCS string data.

The non-predefined LPAD procedure is a program interface intended to be used only when migrating from VS Pascal Release 1 to Release 2, and for no other purposes.

For further information on the LPAD procedure, see "LPAD Procedure" on page 133.

ONERROR Procedure

ONERROR provides a way for you to gain control when run-time errors occur. When a run-time error occurs, the ONERROR procedure is called to perform any necessary action before generating an error message.

You must code the declaration as shown in Figure 289 or use the %INCLUDE member named "ONERROR" provided in the VS Pascal library.

Figure 289 shows the declaration of the ONERROR procedure.

```

(*****
*)
*) RUNTIME ERROR INTERCEPTION ROUTINE
*)
(*****

TYPE
  ERRORTYPE = 1 .. 999;
  ERRORACTIONS = (
    XHALT,
    XPMSG,
    XUMSG,
    XTRACE,
    XDEBUG,
    XDECERR,
    XRESERVED6,
    XRESERVED7,
    XRESERVED8,
    XRESERVED9,
    XRESERVEDA,
    XRESERVEDB,
    XRESERVEDC,
    XRESERVEDD,
    XRESERVEDE,
    XRESERVEDF);

  ERRORSET = SET OF ERRORACTIONS;

PROCEDURE ONERROR(
  CONST FERROR : ERRORTYPE;
  CONST FMODNAME : STRING;
  CONST FPROCNAME : STRING;
  CONST FSTMTNO : INTEGER;
  VAR FRETMSG : STRING;
  VAR FACTION : ERRORSET);
EXTERNAL;
```

Figure 289. Declaration of the ONERROR Procedure

See *VS Pascal Application Programming Guide* for further information about the ONERROR procedure.

PICTURE Function

PICTURE returns the string representation of a real number formatted according to a "picture" specification. The characters that make up the picture specification are similar to those found in PL/I and COBOL. Figure 290 shows the definition of the PICTURE function.

```
FUNCTION PICTURE( CONST p : STRING;
                  r : REAL); STRING(100);
    EXTERNAL;
```

Where	Represents
<i>p</i>	A picture specification
<i>r</i>	A real expression

Figure 290. Definition of the PICTURE Function

You must code the declaration as shown in Figure 290, or use the %INCLUDE member named "CONVERT" provided in the VS Pascal library.

A picture specification consists of two fields: a decimal field and an exponent field. The latter is optional, but the first one is always required.

The decimal field can consist of two subfields: the integer part and the fractional part. The latter is optional, but the first one is always required.

Picture characters can be specified in lowercase. A picture character can be grouped into the following categories:

- Digit and decimal-point specifiers

- D** specifies that the associated position in the data item is to contain a decimal digit.

- V** divides the decimal field into two parts: the integer part and the fractional part. This character specifies that a decimal point is assumed at this position in the associated data item.

Note: However, V does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character. Therefore, an assigned value can be truncated or extended with zeroes at either end. If no V character appears, a V is assumed at the right end of the decimal field.

- Zero-suppression characters

- Z** specifies a conditional digit position in the character string value and causes a leading zero to be replaced with a blank.

- *** specifies a conditional digit position in the character string value and causes a leading zero to be replaced with an asterisk ("*").

Leading zeros are those that occur in the left-most digit positions of the integer part of floating-point numbers.

- Insertion characters

Insertion characters are added into corresponding positions in the output string provided that zero suppression is not taking place. If zeros are being suppressed when an insertion character is encountered, a blank or an asterisk will be inserted in the corresponding place in the output string, depending on whether the zero-suppression character is a Z or an asterisk (*).

- , causes a comma to be inserted into the associated position of the output string.
- . causes a point (.) to be inserted into the associated position of the output string. The character never causes point alignment in the number. That function is served solely by the character V.
- B** causes a blank to be inserted into the associated position of the output string.

- Signs and currency symbol

The sign and currency characters ("S", "+", "-", "\$") can be used in either a static or a drifting manner. The static use specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed.

A drifting character is specified by multiple use of that character in a picture field.

- +** specifies a plus sign character (+) if the number is ≥ 0 ; otherwise it specifies a blank.
- specifies a minus sign character (-) if the number is < 0 ; otherwise it specifies a blank.
- S** specifies a plus sign character (+) if the number is ≥ 0 ; otherwise it specifies a minus sign character (-).
- \$** specifies a dollar sign character (\$).

- Exponent specifiers.

The characters "E" and "K" delimit the exponent field of a picture specification. The exponent field must always be the last field.

- E** specifies that the associated position contains the letter E, which indicates the start of the exponent field.
- K** specifies that the exponent field appears to the right of the associated position. It does not specify a character data item.

See Figure 291 on page 191 for examples of using the PICTURE function.

P	R	PICTURE(P,R)
'99999'	123.0	'00123'
'ZZZZ9'	123.0	' 123'
'****9'	123.0	'**123'
'ZZZZ9'	0.0	' 0'
'ZZZZZ'	0.0	' '
'****9'	0.0	'****0'
'*****'	0.0	'*****'
'S9999'	123.0	'+0123'
'+9999'	123.0	'+0123'
'+9999'	-123.0	' 0123'
'999.99'	-123.456	'001.23'
'999V.99'	123.456	'123.46'
'ZZZ,ZZZ,ZZ9'	123456.0	' 123,456'
'***,***,**9'	123456.0	'***123,456'
'-ZZ,ZZZ,ZZ9'	-123456.0	'- 123,456'
'—,—,—,9'	-123456.0	' -123,456'
'\$**,**,**9V.99'	123456.78	'\$***123,456.78'
'\$\$\$,\$\$\$,\$\$9V.99'	123456.78	' \$123,456.78'
'S9V.9999ES99'	1.23456	'+1.2346E+00'
'S9V.9999KS99'	1.23456	'+1.2346+00'
'-999.999,V99'	1234.567	' 001.234,57'
'-9.999E9'	-1234.567	'-1.235E0'
'98989898989'	123456.0	'1 2 3 4 5 6'
'9.9.9.9.9.9'	12345.0	'0.1.2.3.4.5'
'99999S'	-12345.0	'12345-'
'999+'	-123.45	'123 '
'999+'	+123.45	'123+'
'ZZZ.V99'	0.12	' 12'
'ZZZV.99'	0.12	' .12'
'-9V.999ES9'	1.23E4	' 1.230E+4'
'S9999VESZ9'	-123456.0	'-1235E+ 2'
'-V.999E-99'	123456.0	' .123E 06'

Figure 291. Examples of the PICTURE Function

RPAD Procedure

RPAD pads or truncates a string on the right. It only works with string data; you must use the predefined RPAD procedure to manipulate DBCS string data.

The non-predefined RPAD procedure is a program interface intended to be used only when migrating from VS Pascal Release 1 to Release 2, and for no other purposes.

For further information on the RPAD procedure, see "RPAD Procedure" on page 165.

Chapter 9. Expressions

VS Pascal expressions are similar in function and form to expressions found in other high-level programming languages. Expressions permit you to combine data according to specific computational rules. The type of computation to be performed is directed by four classes of operators. These four classes, according to precedence, are:

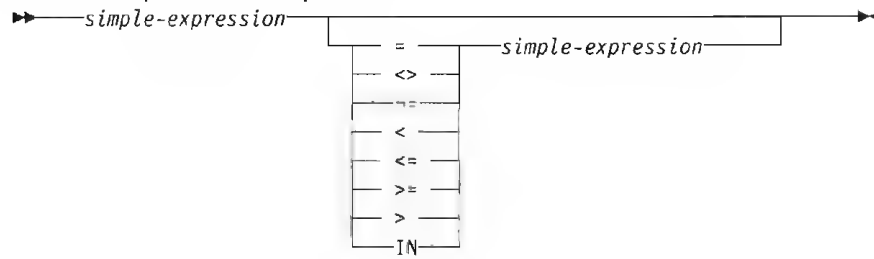
1. The NOT operator (highest)
2. The multiplication operators
3. The addition operators
4. The relational operators (lowest).

An expression is evaluated by applying the operators of highest precedence first, operators of the next precedence second, and so forth. Operators of equal precedence are applied in a left to right order. If an operator has an operand that is a subexpression enclosed within parentheses, the subexpression is evaluated before applying the operator.

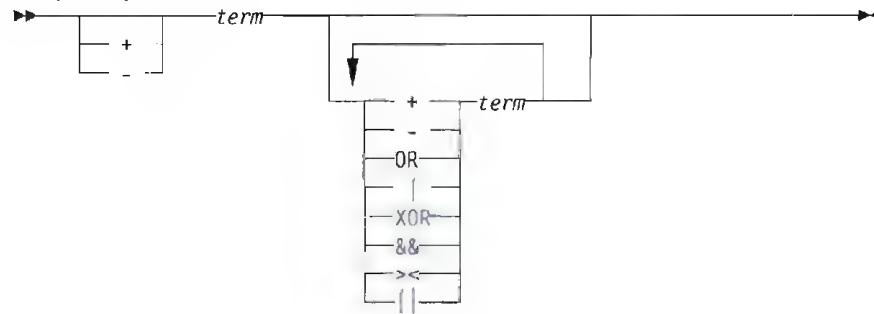
The operands of an expression can be evaluated in either order; therefore, do not expect the left operand of a dyadic operator to be evaluated before the right operand. For example, if one operand contains a function call which modifies a variable used by the other operand, the value used for the variable is unpredictable. The only exception to this evaluation order is with Boolean expressions involving the logical operations of AND (or &) and OR (or |). In these operations, the right operand is not evaluated if the result can be determined from the left operand. See "BOOLEAN Expressions" on page 199 for further information.

Figure 292 on page 195 shows the syntax of VS Pascal expressions.

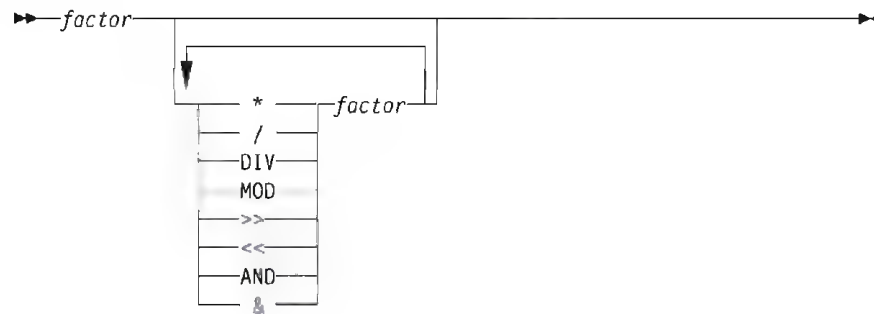
Constant Expression or Expression



Simple Expression



Term



Factor

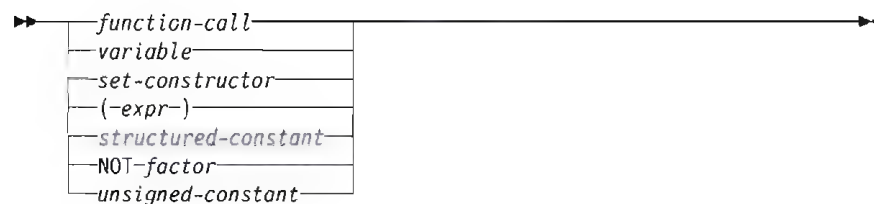


Figure 292. Syntax of VS Pascal Expressions

Note to Figure 292: Because simple-expressions are prefaced with signs, confusion can arise when using signs on the operands. Figure 293 on page 196 shows examples of valid and invalid usage of signs in simple expressions.

```

CONST
  C    = -7;

  X    = C MOD 4;      (* yields 1 *)
  Y    = -7 MOD 4;     (* yields -3 because it is treated *)
                      (* as -(7 MOD 4) *)
  Z    = 7 DIV -4;     (* Error: -4 must be in parentheses *)

```

Figure 293. Examples of Using Signs in Simple Expressions

Figure 294 shows examples of VS Pascal expressions.

Assume the Following Declarations:

```

CONST
  ACME    = 'ACME';

TYPE
  COLORS  = (RED, YELLOW, BLUE);
  SHADES  = SET OF COLORS;
  DAYS    = (SUN, MON, TUES, WED, THUR, FRI, SAT);
  MONTHS  = (JAN, FEB, MAR, APR, MAY, JUN,
             JUL, AUG, SEP, OCT, NOV, DEC);

VAR
  COLOR   : COLORS;
  SHADE   : SHADES;
  BOOL    : BOOLEAN;
  MON     : MONTHS;
  I,
  J       : INTEGER;

```

Factors:

I	Variable
15	Unsigned constant
(I*8+J)	Parentnetical expression
[RED]	Set of one element
[]	Empty set
ODD(I*J)	Function call
NOT BOOL	Complement expression
COLORS(1)	Ordinal conversion
ACME	Constant reference

Terms:

I	Factor
I * J	Multiplication
I DIV J	Integer division
SHADE * [RED]	Set intersection
I & 'FF00'X	Logical AND on integers
BOOL AND ODD(I)	Boolean AND

Figure 294 (Part 1 of 2). Examples of VS Pascal Expressions

Simple Expressions:

I * J	Term
I + J	Addition
I '80000000'X	Logical OR on integers
SHADE + [BLUE]	Set union
- I	Unary minus on an integer
ACME ' TRUCKING'	Catenation

Expressions:

I + J	Simple expression
RED = COLOR	Test for equality
RED IN SHADE	Test for set inclusion

Figure 294 (Part 2 of 2). Examples of VS Pascal Expressions

Operators

Operators are the special symbols (such as "+", for addition) and words (such as DIV, for integer division) that represent algebraic or logical processes to be performed on a value or pair of values. *Operands* are the values manipulated by operators. Operators manipulate operands to produce *results*.

Figures 295, 296, 297, and 298 are organized by precedence, with the NOT operators taking highest precedence. The names for the operators are given according to the main operators in each group.

The NOT Operators

Operator	Operation	Operands	Result
NOT or \neg	Boolean NOT	BOOLEAN	BOOLEAN
NOT or \neg	Logical one's complement	INTEGER	INTEGER
NOT or \neg	Set complement	SET OF t	SET OF t

Figure 295. NOT Operators

The Multiplication Operators

Operator	Operation	Operands	Result
*	Multiplication	INTEGER SHORTREAL REAL Mixed	INTEGER SHORTREAL REAL REAL
*	Set intersection	SET OF t	SET OF t

Figure 296 (Part 1 of 2). Multiplication Operators

Operator	Operation	Operands	Result
/	Real division	INTEGER SHORTREAL REAL Mixed	REAL SHORTREAL REAL REAL
DIV	Integer division	INTEGER	INTEGER
MOD	Modulo	INTEGER	INTEGER
AND or &	Boolean AND	BOOLEAN	BOOLEAN
AND or &	Logical AND	INTEGER	INTEGER
< <	Logical left shift	INTEGER	INTEGER
> >	Logical right shift	INTEGER	INTEGER

Figure 296 (Part 2 of 2). Multiplication Operators

The Addition Operators

Operator	Operation	Operands	Result
+	Addition	INTEGER SHORTREAL REAL Mixed	INTEGER SHORTREAL REAL REAL
+	Set union	SET OF t	SET OF t
+ or	String concatenation	STRING GSTRING	STRING GSTRING
-	Subtraction	INTEGER SHORTREAL REAL Mixed	INTEGER SHORTREAL REAL REAL
-	Set difference	SET OF t	SET OF t
OR or	Boolean OR	BOOLEAN	BOOLEAN
OR or	Logical OR	INTEGER	INTEGER
> < or XOR or &&	Boolean XOR	BOOLEAN	BOOLEAN
> < or XOR or &&	Logical XOR	INTEGER	INTEGER
> < or XOR or &&	Set symmetric difference	SET OF t	SET OF t

Figure 297. Addition Operators

The Relational Operators

Operator	Operation	Operands	Result
=	Compare equal	Any set, scalar, pointer or string	BOOLEAN
< > or ≠	Not equal	Any set, scalar, pointer or string	BOOLEAN
<	Less than	Scalar type or string	BOOLEAN
< =	Compare < or =	Scalar type or string	BOOLEAN
< =	Subset	SET OF t	BOOLEAN
>	Compare greater	Scalar type or string	BOOLEAN
> =	Compare > or =	Scalar type or string	BOOLEAN
> =	Superset	SET OF t	BOOLEAN
IN	Set membership	t and SET OF t	BOOLEAN

Figure 298. Relational Operators

Note to Figure 298: Relational operators manipulate DBCS and mixed strings in a byte-oriented manner.

BOOLEAN Expressions

Pascal assigns higher precedences to logical operators than to relational operators. This means that the expression:

`A < B AND C < D`

is evaluated as:

`(A < (B AND C)) < D.`

Thus, it is advisable to use parentheses when writing expressions of this sort.

VS Pascal optimizes the evaluation of Boolean expressions involving AND and OR so that the right operand of the expression is not evaluated if the result of the operation can be determined by evaluating the left operand. For example, if A, B, and C are Boolean expressions and X is a Boolean variable, the evaluation of:

```
IF A OR B OR C
THEN S;
```

is performed as:

```
IF A
THEN S
ELSE
  IF B
  THEN S
  ELSE
    IF C
    THEN S;
```

The evaluation of:

```
IF A AND B AND C
THEN S;
```

is performed as:

```
IF NOT A
THEN
ELSE
  IF NOT B
  THEN
  ELSE
    IF C
    THEN S;
```

This type of evaluation is called *short circuiting* or *anchor pointing*. The evaluation of the expression is always from left to right.

Usage:

- If you use a function in the right operand of a Boolean expression, the function might not be evaluated. If you rely on a side effect of that function, your program might not work. Note that relying on such side effects denotes an undesirable programming practice; functions should not modify global variables or contain VAR parameters.
- Not all Pascal compilers support this interpretation of Boolean expressions. To ensure portability between VS Pascal and other Pascal implementations, write the compound tests in a form that uses nested IF statements.

Figure 299 demonstrates logic that depends on the conditional evaluation of the right operand of the AND operator. If both operands in the WHILE statement were always evaluated, a NIL pointer checking error would occur when P had the value of NIL.

```
TYPE
  RECPTR = @REC;
  REC = RECORD
    NAME: ALPHA;
    NEXT: RECPTR;
  END;

VAR
  P : RECPTR;
  LNAME : ALPHA;

BEGIN
  .
  .
  WHILE (P<>NIL) AND
    (P@.NAME <> LNAME)
  DO
    P := P@.NEXT;
  .
  .
END;
```

Figure 299. Example of a Boolean Expression

Constant Expressions

Constant expressions are expressions that can be evaluated by the compiler and replaced with a result at compile time. By its nature, a constant expression cannot contain a reference to a variable or to a user-defined function. **Constant expressions can appear in constant declarations, record variant tag lists, and CASE constant lists.**

Figure 300 shows the predefined functions permitted in constant expressions.

Function	Description
ABS	Returns the absolute value of a number
CHR	Returns the EBCDIC character of an expression
FLOAT	Converts an integer to a floating-point value
GSTR	Converts a GCHAR or a DBCS fixed string to a GSTRING
HBOUND	Returns the upper bound of a dimension of an array
HIGHEST	Returns the maximum value of an expression
LBOUND	Returns the lower bound of a dimension of an array
LENGTH	Returns the current length of a string
LOWEST	Returns the minimum value of an expression
MAX	Returns the maximum value of a list of expressions
MAXLENGTH	Returns the maximum length of a string
MIN	Returns the minimum value of a list of expressions
ODD	Returns TRUE if an expression is odd
ORD	Converts a value to an integer, scalar, or pointer expression
PRED	Obtains the predecessor of an expression
ROUND	Converts a floating-point expression to an integer by rounding
SIZEOF	Returns the storage size of a value
SQR	Returns the square of a number
STR	Converts a CHAR or an SBCS fixed string to a STRING
SUCC	Obtains the successor of a type
TRUNC	Converts a floating-point expression to an integer by truncating

Figure 300. Predefined Functions Permitted in Constant Expressions

Figure 301 shows examples of constant expressions.

Constant Expression	Type
ORD('A')	INTEGER
SUCC(CHR(15*16))	CHAR
256 DIV 2	INTEGER

Figure 301 (Part 1 of 2). Examples of Constant Expressions

Constant Expression	Type
'TOKEN' STR(CHR(0))	STRING
'8000'X '0001'X	INTEGER
['0'..'9']	SET OF CHAR
32768*2-1	INTEGER

Figure 301 (Part 2 of 2). Examples of Constant Expressions

Logical Expressions

Many of the integer operators provided in VS Pascal perform logical operations on their operands; the operands are treated as unsigned strings of binary digits instead of signed arithmetic quantities. For example, if the integer value of -1 is an operand of a logical operation, it is viewed as a string of binary digits with a hexadecimal value of 'FFFFFFFF'X.

The logical operations are defined to apply to 32-bit values. Such an operation on a subrange of an integer can possibly yield a result outside the subrange.

Figure 302 shows those operators which perform logical operations on integer operands.

Operator	Operation
& or AND	Performs a bit-wise AND
or OR	Performs a bit-wise inclusive OR
> < or XOR or ^^	Performs a bit-wise exclusive OR
¬ or NOT	Performs a one's complement
< <	Shifts the left operand value left by the number of bits indicated (zeros are shifted in from the right)
> >	Shifts the left operand value right by the number of bits indicated (zeros are shifted in from the left)

Figure 302. Logical Operators for Integer Operands

Figure 303 shows examples of logical operations.

257 & 'FF'X	(* yields 1 *)
2 4 8	(* yields 14 *)
4 << 2	(* yields 16 *)
-4 << 1	(* yields -8 *)
8 >> 1	(* yields 4 *)
-8 >> 1	(* yields '7FFFFFFC'X *)
'FFFF'X >> 3	(* yields '1FFF'X *)
¬1 & 'FF'X	(* yields 'FE'X *)
¬0	(* yields -1 *)
'FF'X ^^ 8	(* yields 'F7'X *)

Figure 303. Examples of Logical Operations

Function Calls

A function returns a value to the invoker. A call to a function passes the actual parameters to the corresponding formal parameters. Parameter compatibility rules are defined in "Routine Parameters" on page 105.

If a user-declared function requires no parameters, an empty set of parentheses can be used on a function call to distinguish the function call from a variable or constant reference. Figure 304 shows the syntax of a function call.

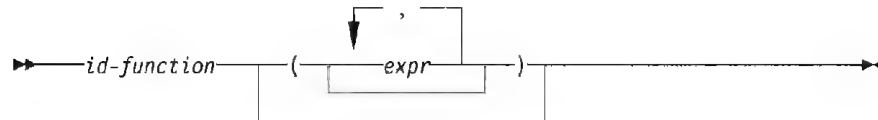


Figure 304. Syntax of a Function Call

Figure 305 shows an example of a function call.

```
VAR
  A,B,C: INTEGER;

FUNCTION SUM(A, B : INTEGER) : INTEGER;
BEGIN
  SUM := A + B;
END;
.
.
BEGIN
  .
  .
  C := SUM(A, B) * 2;
  .
  .
END;
```

Figure 305. Example of a Function Call

Ordinal Conversions

The predefined function ORD converts any ordinal value into an integer. The ordinal conversion functions convert an integer into a specified ordinal type. An integer expression is converted to another ordinal type by enclosing the expression within parentheses and prefixing it with the type identifier of the desired ordinal type. The conversion is performed in such a way as to be the inverse of the ORD function. See "ORD Function" on page 150.

Figure 306 on page 204 shows the syntax of an ordinal conversion.



Figure 306. Syntax of an Ordinal Conversion

By definition, the expression `CHAR(x)` is equivalent to `CHR(x)`, `INTEGER(x)` is equivalent to `x`, and `ORD(type(x))` is equivalent to `x`. Also, `REAL(x)` is equivalent to `FLOAT(x)` for real numbers, and `SHORTREAL(x)` is equivalent to `FLOAT(x)` for shortreal numbers.

Note: Although `REAL` and `SHORTREAL` are not ordinal types, they can be used as ordinal conversion functions.

Figure 307 shows examples of the ordinal conversion function.

```

TYPE
  DAYS = (SUN, MON, TUE, WED,
          THU, FRI, SAT);

.

DAYS(0)      (* IS SUN      *)
DAYS(3)      (* IS WED     *)
DAYS(6)      (* IS SAT     *)
DAYS(7)      (* IS AN ERROR *)
BOOLEAN(0)   (* IS FALSE   *)
BOOLEAN(1)   (* IS TRUE    *)

```

Figure 307. Examples of the Ordinal Conversion Function

Set Constructors

A set constructor is used to compute the value of a `SET` within an expression. Figure 308 shows the syntax of a set constructor.

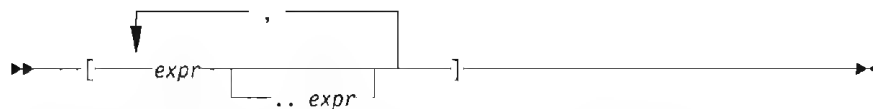


Figure 308. Syntax of a Set Constructor

The set constructor is either a list of expressions separated by commas or groups of expression pairs within square brackets. An expression pair designates that all values from the first expression through the last expression are to be included in the resulting set; if the value of the first expression is greater than the value of the second expression, no values are designated. All expressions must be of compatible types. This type becomes the base scalar type of the set. If the set specifies `INTEGER` expressions, then there is an implementation restriction of 256 elements permitted in the set. Figure 309 on page 205 shows an example of a set constructor.

```

TYPE
  DAYS  = SET OF
    {SUN,MON,TUE,WED,THU,FRI,SAT};
  CHARSET= SET OF CHAR;

VAR
  WORKDAYS,
  WEEKEND:  DAYS;
  NONLETTERS: CHARSET;
  .
  .
BEGIN
  WORKDAYS := [MON..FRI];
  WEEKEND  :=  $\neg$  WORKDAYS;
  NONLETTERS :=
     $\neg$  ['a'..'z','A'..'Z'];
  .
  .
END;

```

Figure 309. Example of a Set Constructor

Chapter 10. Statements

Statements direct the compiler to perform specific operations on data. Statements also control the execution of a program. Statements can be simple, as in an assignment statement, or structured, as in a compound statement including BEGIN and END keywords. VS Pascal statements are similar to those found in most high-level programming languages. Figure 310 shows the syntax of VS Pascal statements.

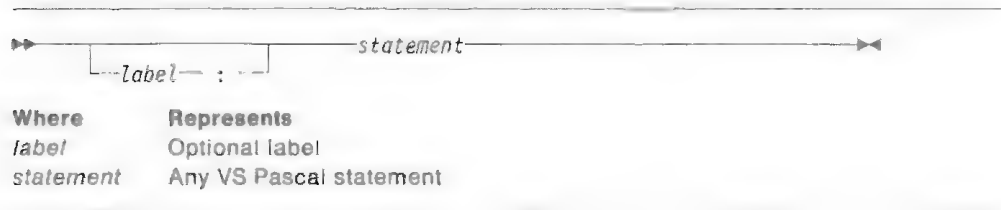


Figure 310. Syntax of VS Pascal Statements

Note to Figure 310: The label must be declared in the routine that contains the statement. The occurrence of a label before a statement is said to "define" the label.

Figure 311 summarizes the VS Pascal statements.

Statement	Description	See Page
ASSERT	Checks for a condition and signals a run-time error if the condition is not met	209
Assignment	Assigns a value to a variable	209
CASE	Permits a program to execute one member of a list of possible statements based upon the evaluation of an expression	211
Compound	Brackets a series of statements that are to be executed sequentially	214
CONTINUE	Causes a jump to the loop-continuation portion of the innermost enclosing FOR, WHILE, or REPEAT	215
Empty	Serves as a place holder and has no effect on the execution of the program	215
FOR	Causes a statement to execute a specified number of times	216
GOTO	Changes the flow of control within a program	219
IF	Specifies that one of two statements is to be executed depending on the evaluation of a Boolean expression	220
LEAVE	Causes an immediate, unconditional exit from the innermost enclosing FOR, WHILE, or REPEAT	222
Procedure call	Invokes a procedure	223

Figure 311 (Part 1 of 2). Summary of VS Pascal Statements

Statement	Description	See Page
REPEAT	Causes statements between the statement delimiters REPEAT and UNTIL to be executed until the control expression is true	223
RETURN	Permits an exit from a procedure or a function	224
WHILE	Causes a statement to be executed as long as a control expression evaluates to true	225
WITH	Simplifies references to a record variable by eliminating an addressing description on every reference to a field	225

Figure 311 (Part 2 of 2). Summary of VS Pascal Statements

VS Pascal Statements

ASSERT Statement

ASSERT checks for a specific condition and signals a run-time error if the condition is not met. The condition is specified by an expression that must evaluate to a Boolean value. If the condition is not true, the error is issued. The compiler might remove the statement from the object program if it can be determined that the assertion is always true. Figure 312 shows the syntax of the ASSERT statement.

<pre> >> ASSERT <i>expr</i> </pre>	
Where	Represents
ASSERT	Statement keyword
<i>expr</i>	A Boolean expression

Figure 312. Syntax of the ASSERT Statement

Figure 313 shows an example of the ASSERT statement.

```

ASSERT A >= B

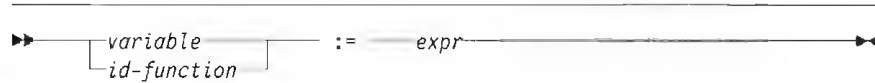
```

Figure 313. Example of the ASSERT Statement

Note to Figure 313: If A is greater than or equal to B, the expression is true and no action is taken; otherwise, an error will be indicated.

Assignment Statement

The assignment statement assigns a value to a variable or a function result. This statement is composed of a reference to a variable followed by the assignment symbol (":="), followed by an expression that, when evaluated, is the new value. The value must be assignment compatible with the variable. The rules for assignment compatibility are given in "Type Compatibility" on page 46. Figure 314 on page 210 shows the syntax of the assignment statement.



Where	Represents
<i>variable</i>	Any variable
<i>id-function</i>	Any user-defined function name
<i>expr</i>	Any expression

Figure 314. Syntax of the Assignment Statement

When you **make** array assignments (assign one array to another array) or record assignments (**assign** one record to another), the entire array or record is assigned.

A result is **returned** from a function by assigning the **result** to the function name **before** leaving the function. See “Function Results” on page 107.

Restrictions:

- You cannot assign a value to a pass-by-CONST parameter.
- You cannot **assign** a value to an object of type FILE or of a type that contains, even indirectly, a file.

Figure 315 shows an example of the assignment statement.

```
TYPE
  CARD = RECORD
    SUIT : (SPADE,
            HEART,
            DIAMOND,
            CLUB);
    RANK : 1..13
  END;

VAR
  X, Y, Z : REAL;

  LETTERS,
  DIGITS,
  ALPHANUMERICS
    : SET OF CHAR;

  I, J, K : INTEGER;

  DECK : ARRAY[ 1..52 ] OF
    CARD;
.
.
BEGIN
  I := 1;
  J := 1;
  K := 1;
  X := Y*Z;
  LETTERS      := [ 'A' .. 'Z' ];
  DIGITS       := [ '0' .. '9' ];
  ALPHANUMERICS := LETTERS + DIGITS;
  DECK[ I ].SUIT := HEART;
  DECK[ J ]      := DECK[ K ];
END.
```

Figure 315. Example of the Assignment Statement

CASE Statement

CASE provides the option of having your program execute one member of a list of possible statements based upon the evaluation of an expression. This statement consists of an expression called the *selector* and a list of statements. The selector must be an ordinal type. Each statement is prefixed with one or more ranges of the same type as the selector; each range is separated by a comma. Each range designates one or more values called *case labels*. Figure 316 on page 212 shows the syntax of the CASE statement.

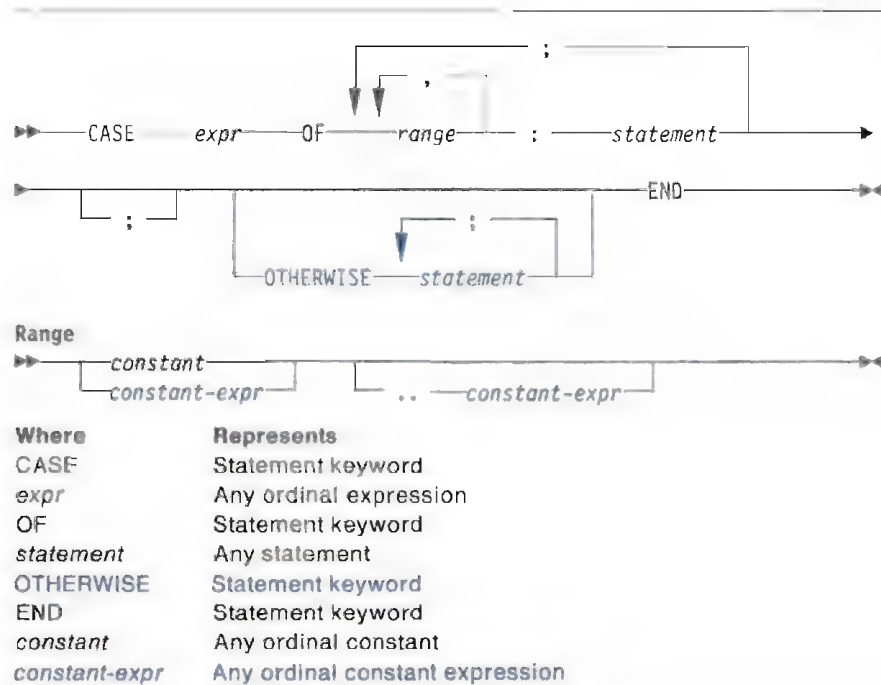


Figure 316. Syntax of the CASE Statement

The `range` values of a CASE statement can be written in any order. However, you cannot designate the same CASE label more than once in the CASE statement.

VS Pascal evaluates the selector and executes the statement whose CASE label equals the value of the selector. When no CASE label equals the value of the selector, the `OTHERWISE` statement, if present, is executed. When no CASE label equals the value of the selector and there is no `OTHERWISE` statement, a run-time error will result when the `%CHECK CASE` compiler directive is enabled. If `%CHECK CASE` is not enabled, VS Pascal can act unpredictably.

Figure 317 on page 213 shows an example of the CASE statement.

```

TYPE
  SHAPE = (TRIANGLE, RECTANGLE,
           SQUARE, CIRCLE);
  COORDINATES =
    RECORD
      X,Y : REAL;
      AREA : REAL;
      CASE S : SHAPE OF
        TRIANGLE:
          (SIDE : REAL;
           BASE : REAL);
        RECTANGLE:
          (SIDEA,SIDEB : REAL);
        SQUARE:
          (EDGE : REAL);
        CIRCLE:
          (RADIUS : REAL);
      END;
    END;
VAR
  COORD : COORDINATES;
BEGIN
  WITH COORD DO
    CASE S OF
      TRIANGLE:
        AREA := 0.5 * SIDE * BASE;
      RECTANGLE:
        AREA := SIDEA * SIDEB;
      SQUARE:
        AREA := SQR(EDGE);
      CIRCLE:
        AREA := 3.14159 * SQR(RADIUS);
    END;
  END;

```

Figure 317. Example of the CASE Statement

Figure 318 on page 214 shows an example of the CASE statement with the OTHERWISE keyword.

```

TYPE
  RANK = (ACE, TWO, THREE, FOUR,
          FIVE, SIX, SEVEN, EIGHT,
          NINE, TEN, JACK, QUEEN,
          KING);
  SJIT = (SPADE, HEART, DIAMOND, CLUB);
  CARD = RECORD
    R : RANK;
    S : SUIT;
  END;
VAR
  POINTS : INTEGER;
  ACARD : CARD;
  .
  .
BEGIN
  CASE ACARD.R OF
    ACE:
      POINTS := 11;
    TWO..TEN:
      POINTS := ORD(ACARD.R)+1;
    OTHERWISE
      POINTS := 10;
  END;

```

Figure 318. Example of the CASE Statement with the OTHERWISE Keyword

Compound Statement

The compound statement serves to bracket a series of statements to be executed sequentially. The reserved words BEGIN and END delimit the statements.

Figure 319 shows the syntax of the compound statement.

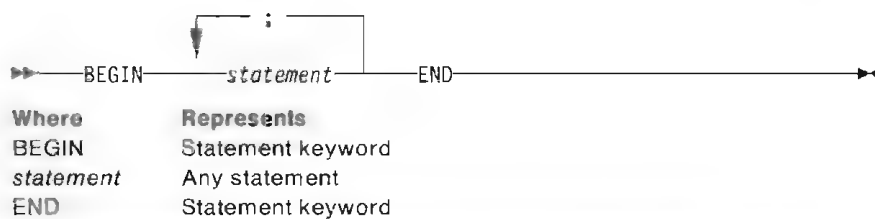


Figure 319 Syntax of the Compound Statement

Figure 320 shows an example of the compound statement.

```

IF A > B THEN
  BEGIN { SWAP A AND B }
    TEMP := A;
    A := B;
    B := TEMP;
  END;

```

Figure 320. Example of the Compound Statement

CONTINUE Statement

CONTINUE causes a jump to the loop-continuation portion of the innermost enclosing FOR, WHILE, or REPEAT statement. In other words, it is a GOTO to the end of the loop. Figure 321 shows the syntax of the CONTINUE statement.




Figure 321. Syntax of the CONTINUE Statement

Figure 322 shows an example of the CONTINUE statement and its equivalent.

This portion of code with a CONTINUE statement:

```
WHILE expr DO
  BEGIN
    .
    .
    IF expr THEN
      CONTINUE;
    .
    .
  END;
```

is equivalent to:

```
WHILE expr DO
  BEGIN
    .
    .
    IF expr THEN
      GOTO label;
    .
    .
  label:      (* Continue jumps to here *)
  END;
```

Figure 322. Example of the CONTINUE Statement and Its Equivalent

Empty Statement

The empty statement is a place holder and has no effect on the execution of the program. This statement is often useful when you want to place a label in the program but do not want it attached to another statement. For example, at the end of a compound statement. Figure 323 shows the syntax of the empty statement.



Figure 323. Syntax of the Empty Statement

The empty statement is also useful for avoiding the ambiguity that arises in nested IF statements. You can force an ELSE clause to be paired with an outer nested IF statement by using an empty statement after an ELSE clause in the inner nested IF statement (see Figure 324 on page 216 for a sample).

```

IF b1 THEN
  IF b2 THEN
    s1
  ELSE
    (*empty statement*)
ELSE
  s2

```

Figure 324. Example of the Empty Statement

FOR Statement

FOR is used to execute a statement a specified number of times. The FOR loop begins with an identifier initialized to the first control expression. With each iteration of the loop, the value of the identifier is either incremented or decremented, depending upon how the statement is coded.

Figure 325 shows the syntax of the FOR statement.

```

▶▶ —FOR— id-var —:=— expr —[TO— | —DOWNTO—] expr —DO— statement —◀◀

```

Where	Represents
FOR	Statement keyword
<i>id-var</i>	An ordinal variable identifier
<i>expr</i>	Any ordinal expression
TO	Increments the value of the control variable
DOWNTO	Decrements the value of the control variable
DO	Statement keyword
<i>statement</i>	Any statement

Figure 325. Syntax of the FOR Statement

To increment the value of the identifier, TO is used between control expressions. The new value of the identifier is computed automatically before the statement is executed. Iterations will continue as long as the value of the identifier is less than or equal to the value of the second control expression.

To decrement the value of the identifier, DOWNTO is used between control expressions. The new value of the identifier is computed automatically before the statement is executed. Iterations will continue as long as the value of the identifier is greater than or equal to the value of the second control expression.

VS Pascal computes the value of the second expression at the beginning of the FOR statement and uses the result for the duration of the statement. Thus, the value of the second control expression is computed once and cannot be changed during the FOR statement. The value of the control variable after the FOR statement is executed is undefined. Do not expect the control variable to contain any particular value.

Restrictions:

- The control variable **must be an** automatic ordinal variable declared in the immediately enclosing routine.
- The control variable cannot be subscripted, field qualified, or referenced through a pointer.
- The executed statement **must not** alter the control variable. If the control variable is altered within the loop, the resultant loop execution is not predictable.
- In the statement contained by the FOR loop, and in any routine declared in the routine immediately enclosing the FOR loop, the control variable cannot be used:
 - In an assignment statement
 - As a control variable of **another** FOR statement
 - As an actual VAR parameter
 - In an input routine (READ, READLN, and so forth).

In the following statement,

```
FOR I := expr1 TO expr2 DO statement
```

I is an automatic scalar variable; *expr1* and *expr2* are scalar expressions that are type-compatible with I, and *statement* is any arbitrary statement.

The compound statement shown in Figure 326 is functionally equivalent to the FOR statement. TEMP1 and TEMP2 are compiler-generated temporary variables.

```
BEGIN
  TEMP1 := expr1;
  TEMP2 := expr2;
  IF TEMP1 <= TEMP2 THEN
    BEGIN
      I := TEMP1;
      REPEAT
        statement;
      IF I = TEMP2 THEN
        LEAVE;
      I := SUC(I);
      UNTIL FALSE; (* forever *)
    END;
  END;
```

Figure 326. Example of the Equivalent of a FOR-TO Statement

In the following statement,

```
FOR I := expr1 DOWNT0 expr2 DO statement
```

I is an automatic scalar variable, *expr1* and *expr2* are scalar expressions that are type compatible with I, and *statement* is any arbitrary statement.

The compound statement shown in Figure 327 on page 218 is functionally equivalent to the FOR statement. TEMP1 and TEMP2 are compiler-generated temporary variables.

```

BEGIN
  TEMP1 := expr1;
  TEMP2 := expr2;
  IF TEMP1 >= TEMP2 THEN
    BEGIN
      I := TEMP1;
      REPEAT
        statement;
      IF I = TEMP2 THEN
        LEAVE;
      I := PRED(I);
    UNTIL FALSE; (* forever *)
  END;
END;

```

Figure 327. Example of the Equivalent of a FOR-DOWNTO Statement

Figure 328 shows examples of the FOR statement.

```

(* Find the maximum integer in an array of integers. *)
MAX := A[1];
LARGEST := 1;
FOR I := 2 TO ASIZE DO
  IF A[I] > MAX THEN
    BEGIN
      LARGEST := I;
      MAX := A[I];
    END;

(* Matrix multiplication: C←A*B *)
FOR I := 1 TO N DO
  FOR J:= 1 TO N DO
    BEGIN
      X := 0.0;
      FOR K := 1 TO N DO
        X := A[I,K] * B[K,J] + X;
      C[I,J] := X;
    END;

(* Sum the hours worked this week *)
SUM := 0;
FOR DAY := MON TO FRI DO
  SUM := SUM + TIMECARD[ DAY ];

```

Figure 328. Examples of the FOR Statement

GOTO Statement

GOTO changes the flow of control within the program. Figure 329 shows the syntax of the GOTO statement.

➤——GOTO—— <i>label</i> ——➤	
Where	Represents
GOTO	Statement keyword
<i>label</i>	A label

Figure 329. Syntax of the GOTO Statement

If a GOTO to a non-local label causes a function to be exited, the function result will not be checked. Figure 330 shows an example of using the GOTO statement to leave a function.

```
LABEL 1;
FUNCTION F : BOOLEAN;
BEGIN
  GOTO 1;
  F := TRUE;
END;

BEGIN
  WRITELN(F);
  1:                                     (* No function check is done *)
END;
```

Figure 330. Example of Using the GOTO Statement to Leave a Function

Restrictions:

- The GOTO statement must be contained by the routine that declared the label.
- You cannot branch into a compound statement from a GOTO statement.
- You cannot branch into the THEN clause or the ELSE clause from a GOTO statement that is outside an IF statement. Further, you cannot branch between the THEN clause and the ELSE clause.
- You cannot branch into a CASE alternative from outside the CASE statement or between CASE alternative statements in the same CASE statement.
- You cannot branch into a FOR, REPEAT, or WHILE loop from a GOTO statement that is not contained within the loop.
- You cannot branch into a WITH statement from a GOTO statement outside of the WITH statement.
- For a GOTO statement that specifies a label defined in an outer routine, the target label cannot be defined within a compound statement or loop.

Figure 331 on page 220 shows examples of valid and invalid GOTO statements.

```

PROCEDURE EXAMPLE;
LABEL
  1, 2, 3, 4

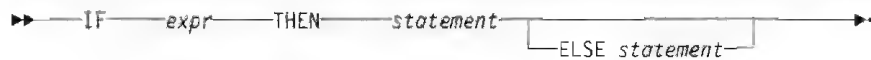
  PROCEDURE INNER;
  BEGIN
    GOTO 4;          (* permitted *)
    GOTO 3;          (* not permitted *)
  END;
BEGIN
  GOTO 3;            (* not permitted *)
  BEGIN
    3:
      GOTO 4;        (* permitted *)
      GOTO 3;        (* permitted *)
    END;
  4: IF EXPR THEN
    1: GOTO 2          (* not permitted *)
    ELSE
    2: GOTO 1;         (* not permitted *)
  END;

```

Figure 331 Example of Valid and Invalid **GOTO** Statements

IF Statement

IF provides the option to specify that one of two statements is to be executed depending on the evaluation of a **Boolean** expression. Each clause contains one statement. Figure 332 shows the syntax of the **IF** statement.



Where	Represents
IF	Statement keyword
<i>expr</i>	Any Boolean expression
THEN	Statement keyword
<i>statement</i>	Any statement
ELSE	Statement keyword

Figure 332. Syntax of the **IF** Statement

The expression must evaluate to a **Boolean** value. If the result of the expression is **TRUE**, the statement in the **THEN** clause is executed. If the expression evaluates to **FALSE** and there is an **ELSE** clause, the statement in the **ELSE** clause is executed; if there is no **ELSE** clause, control passes to the next statement. Figure 333 on page 221 shows examples of simple **IF** statements.

```

IF A <= B THEN
  A := (A + 1.0) / 2.0;

IF ODD(1) THEN
  J := J + 1;
ELSE
  J := J DIV 2 + 1;

```

Figure 333. Examples of Simple IF Statements

VS Pascal always assumes an ELSE clause is paired with the innermost IF statement that does not have an ELSE clause. Nesting an IF statement within an IF statement could be interpreted with two different meanings if only one statement had an ELSE clause.

Figure 334 illustrates this condition, which results in two interpretations:

```

IF b1 THEN IF b2 THEN stmt1 ELSE stmt2

```

Interpretation 1 (assumed by VS Pascal):

```

IF b1 THEN
  BEGIN
    IF b2 THEN
      stmt1
    ELSE
      stmt2
  END

```

Interpretation 2 (incorrect interpretation):

```

IF b1 THEN
  BEGIN
    IF b2 THEN
      stmt1
    END
  ELSE
    stmt2

```

Figure 334. Example of Nested IF Statements

If you prefer the second interpretation, code it as shown, or take advantage of the empty statement, as illustrated in Figure 335.

```

IF b1 THEN
  IF b2 THEN
    stmt1
  ELSE
    (*empty statement*)
ELSE
  stmt2

```

Figure 335. Example of Nested IF Statements with the Empty Statement

LEAVE Statement

LEAVE causes an immediate, unconditional exit from the innermost enclosing FOR, WHILE, or REPEAT loop. Figure 336 shows the syntax of the LEAVE statement.



Figure 336. Syntax of the LEAVE Statement

Figure 337 shows an example of the LEAVE statement.

```
P := FIRST;
WHILE P<>NIL DO
  IF P^.NAME = 'JOE SMITH' THEN
    LEAVE
  ELSE
    P := P^.NEXT;
(*P either points to the desired*)
(*data or is NIL                *)
```

Figure 337. Example of the LEAVE Statement

Figure 338 shows an example of the LEAVE statement and its equivalent.

This portion of code with a LEAVE statement:

```
WHILE expr DO
  BEGIN
    .
    .
    IF expr THEN
      LEAVE;
    .
    .
  END;
```

is equivalent to:

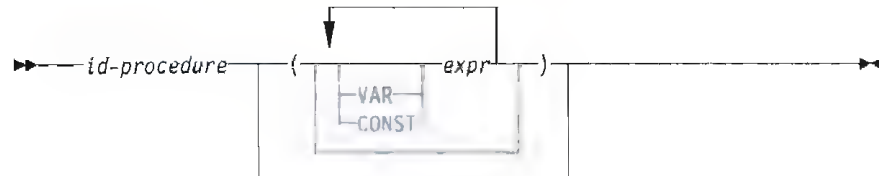
```
WHILE expr DO
  BEGIN
    .
    .
    IF expr THEN
      GOTO label;
    .
    .
  END;
label: ;
```

Figure 338. Example of the LEAVE Statement and Its Equivalent

Procedure Call

A procedure call invokes a procedure. When a procedure is invoked, the actual parameters are substituted for the corresponding formal parameters. For parameter compatibility rules, see “Routine Parameters” on page 105.

If a user-declared procedure requires no parameters, an empty set of parentheses can be used on a procedure call to distinguish the procedure call from a statement. Figure 339 shows the syntax of the procedure call.



Where	Represents
<i>id-procedure</i>	Any predefined or user-defined procedure name
<i>expr</i>	Any expression

Figure 339. Syntax of the Procedure Call

Note to Figure 339: The use of VAR and CONST before expressions is applicable only to GENERIC procedures.

Figure 340 shows an example of procedure calls.

```
TRANSPOSE(MATRIX,
           ROWS,
           COLUMNS);

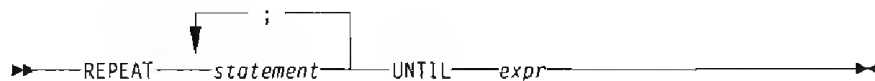
MATRIXADD(A,
          B,
          C,
          N,M);

XYZ(I+J, K*L);
```

Figure 340. Example of Procedure Calls

REPEAT Statement

The statements contained between the statement delimiters REPEAT and UNTIL are executed until the control expression evaluates to true. The control expression must be of type BOOLEAN. Because the termination test is at the end of the loop, the body of the loop is always executed at least once. Because it can contain a list of statements, the REPEAT can act as a compound statement. Figure 341 on page 224 shows the syntax of the REPEAT statement.



Where	Represents
REPEAT	Statement keyword
<i>statement</i>	Any statement
UNTIL	Statement keyword
<i>expr</i>	Any Boolean expression evaluated after each execution of the statement

Figure 341. Syntax of the REPEAT Statement

Figure 342 shows an example of the REPEAT statement, in which the greatest common factor of I and J is stored in I.

```

REPEAT
  K := I MOD J;
  I := J;
  J := K;
UNTIL J = 0;

```

Figure 342. Example of the REPEAT Statement

RETURN Statement

RETURN permits an exit from a procedure or function. This statement is effectively a GOTO to an imaginary label after the last statement within the routine being executed.

Figure 343 shows the syntax of the RETURN statement.



Figure 343. Syntax of the RETURN Statement

When the %CHECK FUNCTION compiler directive is enabled, VS Pascal checks to ensure that a function has been assigned a value before the return from the function. A run-time error message will occur if no value has been assigned.

Figure 344 shows an example of the RETURN statement.

```

PROCEDURE P;
BEGIN
  .
  IF expr THEN RETURN;
  .
END;

```

Figure 344. Example of the RETURN Statement

WHILE Statement

WHILE allows you to specify a statement to be executed as long as a control expression evaluates to true. The condition is tested before the first execution of the statement. Figure 345 shows the syntax of the WHILE statement.

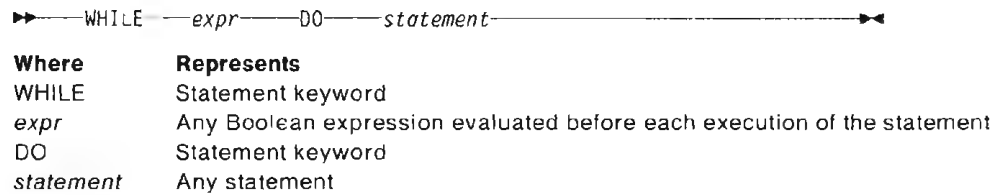


Figure 345. Syntax of the WHILE Statement

The example in Figure 346 computes the decimal size of N assuming $N \geq 1$.

```
I := 0;
J := 1;
WHILE N > 10 DO
  BEGIN
    I := I + 1;
    J := J * 10;
    N := N DIV 10;
  END;
(*I is the power of ten of the *)
(* original N                    *)
(*J is ten to the I power        *)
(*1 <= N <= 9                   *)
```

Figure 346. Example of the WHILE Statement

WITH Statement

WITH simplifies references to a record variable by eliminating an addressing description on every reference to a field. The WITH statement makes the fields of a record available as if the fields were variables within the nested statement. Figure 347 shows the syntax of the WITH statement.

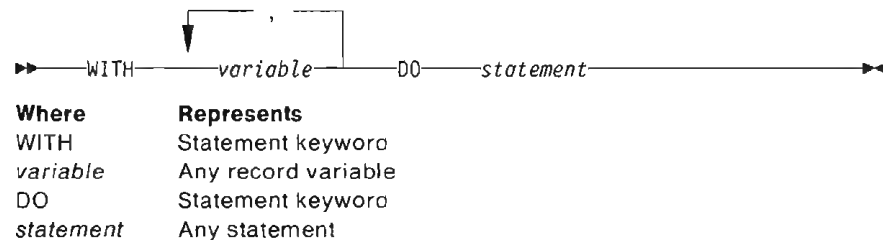


Figure 347. Syntax of the WITH Statement

Figure 348 on page 226 shows an example of the WITH statement.

```

TYPE
  EMPLOYEE =
    RECORD
      NAME   : STRING(20);
      BADGE  : 0..999999;
      SALARY : INTEGER;
      ID     : 0..999999;
    END;

VAR
  FATHER : @ EMPLOYEE;
BEGIN
  NEW(FATHER);
  WITH FATHER@ DO
    BEGIN
      NAME := 'LUIS TAN';
      BADGE := 666666;
      SALARY := STARTING;
      ID := BADGE;
    END;
  END;

The WITH statement is equivalent to:

BEGIN
  FATHER@.NAME := 'LUIS TAN';
  FATHER@.BADGE := 666666;
  FATHER@.SALARY := STARTING;
  FATHER@.ID := FATHER@.BADGE;
END;

```

Figure 348. Example of the WITH Statement

Note to Figure 348: The variable FATHER is a pointer to a dynamic variable of type EMPLOYEE; thus, FATHER must be dereferenced to access the EMPLOYEE record.

The WITH statement effectively computes the address of a record variable upon executing the statement. Any modification to a variable that changes the address computation will not be reflected in the precomputed address during the execution of the WITH statement. Figure 349 illustrates this point.

```

VAR
  A : ARRAY[ 1..10 ] OF
    RECORD
      FIELD : INTEGER;
    END;
BEGIN
  I := 1;
  WITH A[ I ] DO
    BEGIN
      K := FIELD;      (*K:=A[1].FIELD*)
      I := 2;
      K := FIELD;      (*K:=A[1].FIELD*)
    END;
  END;
END;

```

Figure 349. Example of WITH Statement Evaluation

The comma notation of a WITH statement is an abbreviation of nested WITH statements. The names within a WITH statement are scoped such that the last WITH statement will take precedence. A local variable with the same name as a field of a record becomes unavailable in a WITH statement that specifies the record. Figure 350 shows an example of a nested WITH statement and identifier scoping.

```
VAR
  V : RECORD
    V1 : RECORD
      A : REAL;
    END;
    V2 : INTEGER;
    A : INTEGER;
  END;
  A : CHAR;
  .
  .
BEGIN
  WITH V,V1 DO
    BEGIN
      V2 := 1;      (*V.V2 := 1    *)
      A  := 1.0;    (*V.V1.A := 1.0 *)
      V.A := 1;     (*V.A  := 1    *)
                      (*CHAR A is not *)
                      (*available here*)
    END;
    A := 'A';       (*CHAR A is now *)
                      (*available   *)
  END;
```

Figure 350. Example of Nested WITH Statement and Identifier Scoping

Chapter 11. Compiler Directives

Chapter 11. Compiler Directives

VS Pascal compiler directives enable or disable a number of compile-time options and features. The VS Pascal compiler recognizes these directives by the % symbol that precedes them.

The compiler checks these directives for proper syntax. For directives that take suboptions, the compiler reads and validates the option list; it ignores everything past the end of the option list. Thus, you can insert a comment without comment delimiters after an option list. For example,

```
%CHECK SUBRANGE OFF      This turns subrange checking off
```

compiles without error, and "This turns subrange checking off" is treated as a comment.

Directives that take an optional character string accept *everything* on that line as part of the directive. Therefore, use comments on these lines only if you intend for them to be part of the directive. For example, use

```
%TITLE The following are TYPE declarations. (* As of 5/17 *)
```

only if you want the comment "(* As of 5/17 *)" to appear in your title.

Figure 351 summarizes VS Pascal compiler directives by function. Detailed explanations are given in alphabetic order, starting on page 231.

Function	Directive	Associated Compiler Option	Description	Default	See Page
Compiler Listing Format	%CPAGE	SOURCE	Forces a conditional page break	—	232
	%LIST	LIST	Controls whether the pseudo-assembler listing is included	On	234
	%PAGE	SOURCE	Forces a skip to the next page of the listing	—	235
	%PRINT	SOURCE	Controls whether or not source statements are printed	On	235
	%SKIP	SOURCE	Inserts one or more blank lines in the listing	1	236
	%SPACE	SOURCE	Inserts one or more blank lines in the listing	1	236
	%TITLE	SOURCE	Forces a page break and prints a title at the top of the following page	No title	237

Figure 351 (Part 1 of 2). Summary of VS Pascal Compiler Directives

Function	Directive	Associated Compiler Option	Description	Default	See Page
Compiler Instructions	%CHECK	CHECK	Controls run-time checking features	On	231
	%INCLUDE	LIB	Specifies that source from a library file is to be inserted	—	233
	%MARGINS	—	Redefines the left and right margins of the compiler input	—	235
	%UHEADER	HEADER	Places a user header after every routine header	No user header	237
	%WRITE	WRITE	Writes a message to the terminal during the compilation of the unit	No message	240
Conditional Compilation	%ENDSELECT	—	Marks the end of a section of code to be selectively compiled	—	233
	%SELECT	—	Marks the start of a section of code to be selectively compiled	—	236
	%WHEN	CONDPARM	Controls which sections of code are selectively compiled	—	238

Figure 351 (Part 2 of 2). Summary of VS Pascal Compiler Directives

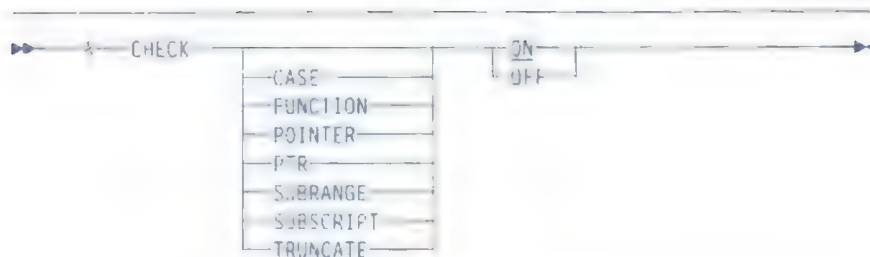
VS Pascal Compiler Directives

%CHECK Directive

%CHECK controls the run-time checking features of VS Pascal. You can enable checking for part or all of a program.

Note: The %CHECK directive works only when the CHECK compile-time option is in effect. When the CHECK compile-time option is on, all checks are applicable unless a %CHECK directive specifies otherwise. For example, when the CHECK option is on and %CHECK POINTER OFF is specified, pointer references will not be checked.

Figure 352 on page 232 shows the syntax of the %CHECK directive.



Where	Instructs the Compiler to
CASE	Flag the value of a CASE statement selector that is not equal to any of the CASE labels
FUNCTION	Flag the lack of an assignment of a value to a function before exiting from the function
POINTER	Flag the dereferencing of a pointer whose value is NIL
PTR	Flag the dereferencing of a pointer whose value is NIL (synonymous with POINTER)
SUBRANGE	Flag the assignment of a value that is not in the proper range for the target variable or parameter; also flag the values passed to some predefined routines if they are not in certain ranges
SUBSCRIPT	Flag the use of a subscript out of range for the array
TRUNCATE	Flag the value of a string that will not fit into the target string on an assignment
ON	Turn checking on; this is the default
OFF	Turn checking off

Figure 352. Syntax of the %CHECK Directive

%CHECK, like all compiler directives, is an instruction to the compiler. Its effect depends only on where it appears in the text. It is not subject to any structuring established by the program.

If %CHECK SUBRANGE is enabled, the following predefined routines will have the following values checked:

- CHR will be flagged if the actual parameter does not yield a valid EBCDIC character.
- WRITE and WRITELN will be flagged if the field widths specified are less than 1 (in LANGVL(ANSI83) only).

%CPAGE Directive

%CPAGE forces a page eject if less than a specified number of lines are left on the current page of the output listing. Use this directive to ensure a unit of code is not split across two pages.

Note: The %CPAGE directive works only when the SOURCE compile-time option is in effect.

Figure 353 shows the syntax of the %CPAGE directive.



Where	Represents
<i>unsigned-integer</i>	The number of lines that must remain on the page for a page eject not to take place

Figure 353. Syntax of the %CPAGE Directive

Figure 354 on page 233 shows an example of the %CPAGE directive.

```
%CPAGE 30
```

Figure 354. Example of the %CPAGE Directive

%ENDSELECT Directive

%ENDSELECT marks the end of a section to be selectively compiled. The section begins with a %SELECT directive. Figure 355 shows the syntax of the %ENDSELECT directive.

```
►► %ENDSELECT ◄◄
```

Figure 355 Syntax of the %ENDSELECT Directive

See “%WHEN Directive” on page 238 to see how %ENDSELECT is used in conjunction with %SELECT and %WHEN.

%INCLUDE Directive

%INCLUDE instructs the compiler to start reading text from the specified library file. After the compiler has read the entire member, the compiler resumes reading from the line immediately following the %INCLUDE directive.

Figure 356 shows the syntax of the %INCLUDE directive.

```
►► %INCLUDE library-name (member-name) ◄◄
           └── member-name ─┘
```

Where

library-name (member-name)

member-name

References

A library file and a specific member in the file. Under VM/CMS and MVS, the specified library name is the ddname of a partitioned data set (which might be concatenated).

The member of the first library in the search order that has a member with the name *member-name*.

Figure 356 Syntax of the %INCLUDE Directive

When a %INCLUDE is encountered, an implicit %MARGINS 1 72 is performed for the member being included.

Figure 357 on page 234 shows an example of the %INCLUDE directive.

```
PROGRAM ABC;  
CONST  
  %INCLUDE CONSTS  
TYPE  
  %INCLUDE TYPES  
VAR  
  %INCLUDE VARS  
%INCLUDE LIB1(PROCS)  
BEGIN  
  .  
  .  
  .  
END.
```

Figure 357. Example of the %INCLUDE Directive

See *VS Pascal Application Programming Guide* for compile-time considerations for programs that use %INCLUDE.

%LIST Directive

%LIST enables and disables the pseudo-assembler listing of the VS Pascal compiler.

Note: The %LIST directive works only when the LIST compile-time option is in effect.

Figure 358 shows the syntax of the %LIST directive.



Where	Instructs the Compiler to
ON	List pseudo-assembler code; this is the default
OFF	Cease listing pseudo-assembler code

Figure 358. Syntax of the %LIST Directive

Using %LIST for a Small Section of a Unit: If you want to view the pseudo-assembler listing for only a small section of a unit, follow these steps:

1. Insert
 %LIST OFF
 at the beginning of the unit.
2. Insert
 %LIST ON
 at the beginning of each section of code for which you want an assembler listing.
3. At the end of each code section insert:
 %LIST OFF
4. Compile the unit with the LIST compile-time option.

%MARGINS Directive

%MARGINS redefines the left and right margins of the compiler input. The compiler skips all characters that lie outside the margins. Figure 359 shows the syntax of the %MARGINS directive.

```

▶ — %MARGINS integer1 integer2 — ▶

```

Where **Represents**

integer1 An unsigned integer to indicate the new left margin

integer2 An unsigned integer to indicate the new right margin

Figure 359 Syntax of the %MARGINS Directive

If a %MARGINS directive appears in a library member being inserted by a %INCLUDE directive, the new margins are used only in that library member. When the compiler reaches the end of the library member, the margin settings revert to their previous values.

%PAGE Directive

%PAGE forces a skip to the next page on the output listing of the source program.

Note: The %PAGE directive works only when the SOURCE compile-time option is in effect.

Figure 360 shows the syntax of the %PAGE directive.

```

▶ — %PAGE — ▶

```

Figure 360 Syntax of the %PAGE Directive

%PRINT Directive

%PRINT controls whether or not source statements are printed in the output listing. The compiler prints the line containing the %PRINT OFF directive, then stops printing until a %PRINT or %PRINT ON directive is processed.

Note: The %PRINT directive works only when the SOURCE compile-time option is in effect.

Figure 361 shows the syntax of the %PRINT directive.

```

▶ — %PRINT —▶
      |
      +-- ON
      |
      +-- OFF

```

Where **Instructs the Compiler to**

ON Print source statements, this is the default

OFF Cease printing source statements

Figure 361 Syntax of the %PRINT Directive

%SELECT Directive

%SELECT marks the beginning of a section of code to be selectively compiled. The selection is based on the values of parameters passed with the CONDPARM compile-time option. Figure 362 shows the syntax of the %SELECT directive.



Figure 362. Syntax of the %SELECT Directive

Restriction: You cannot nest %SELECT – %ENDSELECT groups.

See “%WHEN Directive” on page 238 for an explanation of how %SELECT works in conjunction with %WHEN and %ENDSELECT.

%SKIP Directive

%SKIP inserts one or more blank lines in the source listing.

Note: The %SKIP directive works only when the SOURCE compile-time option is in effect.

Figure 363 shows the syntax of the %SKIP directive.



Where	Represents
<i>integer</i>	An unsigned integer to indicate the number of blank lines; the default is to skip one line


Figure 363. Syntax of the %SKIP Directive

%SPACE Directive

%SPACE inserts one or more blank lines in the source listing, just as the %SKIP directive does.

Note: The %SPACE directive works only when the SOURCE compile-time option is in effect.

Figure 364 shows the syntax of the %SPACE directive.



Where	Represents
<i>integer</i>	An unsigned integer to indicate the number of blank lines; the default is to skip one line

Figure 364. Syntax of the %SPACE Directive

%TITLE Directive

%TITLE places a title in the listing and causes a page skip. The title prints exactly as you type it in, with no change from lowercase to uppercase. If the GRAPHIC compile-time option is in effect, DBCS portions of the directive are checked for validity.

Note: The %TITLE directive works only when the SOURCE compile-time option is in effect.

Figure 365 shows the syntax of the %TITLE directive.

►► **%TITLE** *character-string* ►►

Where	Represents
<i>character-string</i>	Any character string, the default is no title

Figure 365. Syntax of the %TITLE Directive

%UHEADER Directive

%UHEADER places a character string (a "user header") in an optional field following the routine header in the generated object code. The header will appear exactly as you type it in, with no change from lowercase to uppercase. DBCS portions of the %UHEADER directive are never checked for validity.

Note: The %UHEADER directive works only when the HEADER compile-time option is in effect.

Figure 366 shows the syntax of the %UHEADER directive.

►► **%UHEADER** *character-string* ►►

Where	Represents
<i>character-string</i>	Any character string

Figure 366. Syntax of the %UHEADER Directive

The compiler always uses the last %UHEADER directive in effect before the declaration of a routine. If you want a user header on your main program, place %UHEADER ahead of the reserved word PROGRAM.

If you specify %UHEADER with no character string following, no user header will appear in the routine header.

Figure 367 on page 238 shows an example of the %UHEADER directive.


```
%UHEADER COPYRIGHT 1981, 1987 BY IBM; Object Code Only
PROGRAM P;
PROCEDURE P;
BEGIN
END;    (* P has 'COPYRIGHT 1981, 1987; Object Code Only' in user header *)

%UHEADER COPYRIGHT 1983 BY IBM
PROCEDURE Q;
BEGIN
END;    (* Q has 'COPYRIGHT 1983 BY IBM' in user header *)

PROCEDURE R;
BEGIN
END;    (* R has 'COPYRIGHT 1983 BY IBM' in user header *)

PROCEDURE S;
BEGIN
    %UHEADER COPYRIGHT 1987 BY IBM
    %UHEADER
END;    (* S has 'COPYRIGHT 1983 BY IBM' in user header *)

PROCEDURE T;
BEGIN
END;    (* T has no user header *)

%JHEADER COPYRIGHT 1982 BY IBM
BEGIN
END.    (* Main program has 'COPYRIGHT 1981, 1987  *
        (* BY IBM; Object Code Only' in user header *)
```

Figure 367. Example of the %UHEADER Directive

% WHEN Directive

%WHEN determines which block of code delimited by the %SELECT and %ENDSELECT directives is to be compiled. Figure 368 shows the syntax of the %WHEN directive.

► **% WHEN** *Boolean-expression* ◀

Where*Boolean-expression***Represents**

A Boolean expression similar to any valid VS Pascal Boolean expression except that it may not contain VS Pascal identifiers, and may contain conditional parameters passed to the compiler with the CONDPARM compile-time option

Figure 368. Syntax of the %WHEN Directive

Inside each portion of code enclosed by a %SELECT — %ENDSELECT pair, you can include any number of %WHEN directives, each followed by source code. During compilation, the Boolean expression associated with a %WHEN statement is evaluated using the values specified with the CONDPARM compile-time option. If the expression evaluates to TRUE, the source code following the %WHEN is compiled until another %WHEN or a %ENDSELECT is encountered.

If the expression evaluates to FALSE, the compiler bypasses all statements until another %WHEN or a %ENDSELECT is encountered. Only those statements following the first TRUE %WHEN directive are compiled.

Figure 369 shows an example of conditional compilation.

```

PROGRAM ADD_TWO_NUMBERS;
VAR
  X,Y,Z: INTEGER;

BEGIN
  READLN(X);                (*Statement 1*)
  READLN(Y);                (*Statement 2*)

  %SELECT
  %WHEN TRACE = 'XONLY'
    WRITELN('The variable X has value ',X);    (*Statement 3*)
  %WHEN TRACE = 'YONLY'
    WRITELN('The variable Y has value ',Y);    (*Statement 4*)
  %WHEN TRACE = 'XANDY'
    WRITELN('The variable X has value ',X);    (*Statement 5*)
    WRITELN('and the variable Y has value ',Y); (*Statement 6*)
  %ENDSELECT

  X := X + Y                (*Statement 7*)
  WRITELN('The sum of two numbers is ',Z);    (*Statement 8*)
END;

```

When you apply various settings of the CONDPARM compile-time option to this example, you receive these results:

CONDPARM Option	Statements Compiled
CONDPARM(Trace = 'XONLY')	1, 2, 3, 7, 8
CONDPARM(Trace = 'YONLY')	1, 2, 4, 7, 8
CONDPARM(Trace = 'XANDY')	1, 2, 5, 6, 7, 8
CONDPARM(Trace = 'DUMMY')	1, 2, 7, 8

Figure 369. Example of Conditional Compilation

Notes:

- %SELECT can be thought of as starting a comment that is closed by a %WHEN directive that evaluates to TRUE or by a %ENDSELECT directive. The first %WHEN directive following a block of compiled code can be thought of as starting a comment that is closed by the %ENDSELECT directive.
- Only code following the first true %WHEN directive will be compiled. When the end of this %WHEN group is found, nothing will be compiled until a %ENDSELECT directive is found.
- If you want an OTHERWISE capacity, use a %WHEN TRUE as the last %WHEN directive of the %SELECT—%ENDSELECT group.
- If a %WHEN references a conditional parameter not specified with CONDPARM, the compiler assumes a blank. You can use this as a default.
- If you place any statements between %SELECT and the first %WHEN, the compiler ignores them and issues a warning message.

%WRITE Directive

%WRITE allows a message to be written to the terminal at a specified location in the program during compilation. If the GRAPHIC compile-time option is in effect, DBCS portions of the directive are checked for validity.

Note: The %WRITE directive works only when the WRITE compile-time option is in effect.

Figure 370 shows the syntax of the %WRITE directive.

►► **%** **WRITE** *character-string* ►►

Where	Represents
<i>character-string</i>	Any character string

Figure 370. Syntax of the %WRITE Directive

Appendix A. Summary of Changes

Appendix A. Summary of Changes

VS Pascal Release 2 provides the following additions and enhancements to VS Pascal Release 1 that are directly related to defining the VS Pascal programming language and its syntax.

- **System Flexibility**

VS Pascal now allows communication with other IBM licensed programs, such as IMS, using the `GENERIC` routine directive.

- **Compiler Features**

Users now have the option to:

- *Compile only selected portions of a source program.* This "conditional compilation" feature can simplify debugging and help support multiple operating environments.
- *Place headers in generated code.* Headers include the name of the compiled routine, the compiler name, and the date and time of compilation. Users can also insert a customized header after the compiler header.

- **Error Handling**

VS Pascal Release 2 now:

- *Checks compiler directives for syntax, semantic, and limit errors.*
- *Finishes printing its cross-reference and statistics listing when it encounters a severe error.* Previously, severe errors caused the listing to abort immediately.
- *Flags invalid compile-time options, syntax, and suboptions that were previously ignored.*

- **Storage Considerations**

VS Pascal allows users to tune programs by adding multiple heap support. This helps alleviate storage fragmentation problems. A component of a large, multicomponent program can now create and manage its own heap independent of heaps associated with other components.

- **Double-Byte Character Set (DBCS) Data**

Among many new DBCS features, Release 2 supports:

- *A predefined scalar data type, `GCHAR`, which represents one DBCS character.*
- *A predefined structured data type, `GSTRING`, which represents a DBCS string.*
- *Hexadecimal graphic data.*

Existing string manipulation routines were revised for DBCS support, and special DBCS routines for handling mixed strings were added:

- *Many existing string routines now work with `GSTRING` data in a character-oriented manner (two bytes at a time).*
- *New string routines work with `SBCS` and `DBCS STRING` data in a character-oriented manner (one byte at a time for `SBCS` data, and two bytes at a time for `DBCS` data).*

- **Proposed ANSI/IEEE Extended Pascal**

VS Pascal Release 2 adds support for:

- *The EPSREAL predefined constant*
- *The MAXCHAR predefined constant*
- *A plus sign (+) for string concatenation (||)*
- *A set symmetric difference operator (> <).*

- **National Language Support**

VS Pascal Release 2 also:

- *Adopts the Syntactic Graphic Character Set (GCSGID 640) as its standard character set.* VS Pascal programs never require characters outside this set, establishing a standard that makes programs easily transportable among different sites.
- *Allows customization of character translation and uppercase tables during installation.* This eases compiler recognition of tokens and characters due to national programming standards and allows the creation of uppercase rules.

Appendix B. Predefined Identifiers

Appendix B. Predefined Identifiers

A predefined identifier is the name of a constant, type, variable or routine that is predefined in VS Pascal. The name is declared in every unit prior to the start of your program. You can redefine the name if you wish; however, it is better to use the name according to its predefined meaning.

Identifier	Form	Description
ABS	Function	Computes the absolute value of a number
ADDR	Function	Returns the address of a variable
ALFA	Type	Array of 8 characters, indexed 1..ALFALEN
ALFALEN	Constant	HBOUND of type ALFA, value is 8
ALPHA	Type	Array of 16 characters, indexed 1..ALPHALEN
ALPHALEN	Constant	HBOUND of type ALPHA, value is 16
ARCTAN	Function	Returns the arctangent of the argument
BOOLEAN	Type	Data type composed of the values FALSE and TRUE
CHAR	Type	Character data type
CHR	Function	Converts an integer to a character value
CLOCK	Function	Returns the number of microseconds of execution
CLOSE	Procedure	Closes a file
COLS	Function	Returns current column on output line
COMPRESS	Function	Replaces multiple blanks in a string with one blank
COS	Function	Returns the cosine of the argument
DATETIME	Procedure	Returns the current date and time of day
DELETE	Function	Returns a string with a portion removed
DISPOSE	Procedure	Deallocates a dynamic variable
DISPOSEHEAP	Procedure	Deallocates a heap
EOF	Function	Test file for end of file condition
EOLN	Function	Test file for end of line condition
EPSREAL	Constant	Real constant such that $1.0 + \text{EPSREAL} > 1.0$
EXP	Function	Returns the base of the natural log (e) raised to the power of the argument
FALSE	Constant	Constant of type BOOLEAN, $\text{FALSE} < \text{TRUE}$
FLOAT	Function	Converts an integer to a floating-point value
GCHAR	Type	DBCS character data type

Figure 371 (Part 1 of 4). Predefined Identifiers

Identifier	Form	Description
GET	Procedure	Advances file pointer to next element of input file
GSTR	Function	Converts a GCHAR or DBCS fixed string to a GSTRING
GSTRING	Type	An array of DBCS characters whose length varies during execution up to a maximum length
GTOSTR	Function	Converts a GSTRING to a STRING
HALT	Procedure	Halts the programs execution
HBOUND	Function	Returns the upper bound of an array
HIGHEST	Function	Returns the maximum value of an ordinal type
INDEX	Function	Finds the first occurrence of one string in another
INPUT	Variable	Default input file
INTEGER	Type	Integer data type
LBOUND	Function	Returns the lower bound of an array
LENGTH	Function	Returns the current length of a string
LN	Function	Returns the natural logarithm of the argument
LOWEST	Function	Returns the minimum value of an ordinal type
LPAD	Procedure	Pads strings on the left
LTOKEN	Procedure	Extracts tokens from a string
LTRIM	Function	Returns a string with leading blanks removed
MARK	Procedure	Creates a new subheap
MAX	Function	Returns the maximum value of a list of scalars
MAXCHAR	Constant	Constant equal to 'FF'XC
MAXINT	Constant	Maximum value of type INTEGER
MAXLENGTH	Function	Returns the maximum length of a string
MAXREAL	Constant	Maximum value of type REAL
MCOMPRESS	Function	Replaces multiple blanks in a mixed string with one blank
MDELETE	Function	Returns a mixed string with a portion removed
MIN	Function	Returns the minimum value of a list of scalars
MINDEX	Function	Finds the first occurrence of one mixed string in another
MININT	Constant	Minimum value of type INTEGER
MINREAL	Constant	Minimum value of type REAL (smallest non-zero floating-point number)
MLENGTH	Function	Returns the length of a mixed string
MLTRIM	Function	Returns a mixed string with the leading blanks removed

Figure 3/1 (Part 2 of 4). Predefined Identifiers

Identifier	Form	Description
MRINDEX	Function	Locates the last occurrence of one mixed string in another
MSUBSTR	Function	Returns a specific portion of a mixed string
MTRIM	Function	Returns a mixed string with trailing blanks removed
NEW	Procedure	Allocates a dynamic variable from the current heap
NEWHEAP	Procedure	Allocates a new heap
ODD	Function	Returns TRUE if integer argument is odd
ORD	Function	Converts an ordinal value or pointer to an integer
OUTPUT	Variable	Default output file
PACK	Procedure	Copies an array to a packed array
PAGE	Procedure	Skips to the top of the next page
PARMS	Function	Returns the system dependent invocation parameters
PDSIN	Procedure	Opens a file for input from a partitioned data set
PDSOUT	Procedure	Opens a file for output to a partitioned data set
PRED	Function	Obtains the predecessor of an ordinal type
PUT	Procedure	Advances file pointer to next element of output file
QUERYHEAP	Procedure	Identifies the current heap
RANDOM	Function	Returns a pseudo-random number
READ	Procedure	Reads data from a file
READLN	Procedure	Reads one line of a text file
READSTR	Procedure	Converts a string to values assigned to variables
REAL	Type	Floating-point represented in 370 long floating-point
RELEASE	Procedure	Releases storage in a subheap
RESET	Procedure	Opens a file for input
RETCODE	Procedure	Sets the system dependent return code
REWRITE	Procedure	Opens a file for output
RINDEX	Function	Finds the last occurrence of one string in another
ROUND	Function	Converts a floating-point number to an integer by rounding
RPAD	Procedure	Pads a string on the right
SEEK	Procedure	Positions an opened file at a specific record
SHORTREAL	Type	Floating-point represented in 370 short floating-point
SIN	Function	Returns the sine of the argument
SIZEOF	Function	Returns the storage size of a variable or type

Figure 371 (Part 3 of 4). Predefined Identifiers

Identifier	Form	Description
SQR	Function	Returns the square of the argument
SQRT	Function	Returns the square root of the argument
STOGSTR	Function	Converts a STRING to a CSTRING
STR	Function	Converts a CHAR or SBCS fixed string to a STRING
STRING	Type	An array of characters whose length varies during execution up to a maximum length
STRINGPTR	Type	A type for dynamically allocated strings of an execution determined length
SUBSTR	Function	Returns a portion of a string
SUCC	Function	Obtains the successor of an ordinal type
TERMIN	Procedure	Opens a file for input from the terminal
TERMOUT	Procedure	Opens a file for output to the terminal
TEXT	Type	File of CHAR
TOKEN	Procedure	Extracts tokens from a string
TRACE	Procedure	Writes the routine return stack
TRIM	Function	Returns a string with trailing blanks removed
TRUE	Constant	Constant of type BOOLEAN, TRUE > FALSE
TRUNC	Function	Converts a floating-point number to an integer by truncating
UNPACK	Procedure	Copies a packed array to an array
UPDATE	Procedure	Opens a file for both input and output
USEHEAP	Procedure	Changes the current heap
WRITE	Procedure	Writes data to a file
WRITELN	Procedure	Writes one line to a text file
WRITESTR	Procedure	Converts a series of expressions into a string

Figure 371 (Part 4 of 4). Predefined Identifiers

Appendix C. Options for Opening Files

Appendix C. Options for Opening Files

All VS Pascal procedures that open files are defined with an optional string parameter that contains options pertaining to the file being opened. File opening options determine how the file will be opened and what attributes it will have.

These options are valid for the following I/O routines:

PDSIN
PDSOUT
RESET
REWRITE
TERMIN
TERMOUT
UPDATE

Not all of the options apply to all open procedures. If an invalid option is specified for a procedure, the option will be ignored.

Figure 372 shows the syntax of the options string for opening files.

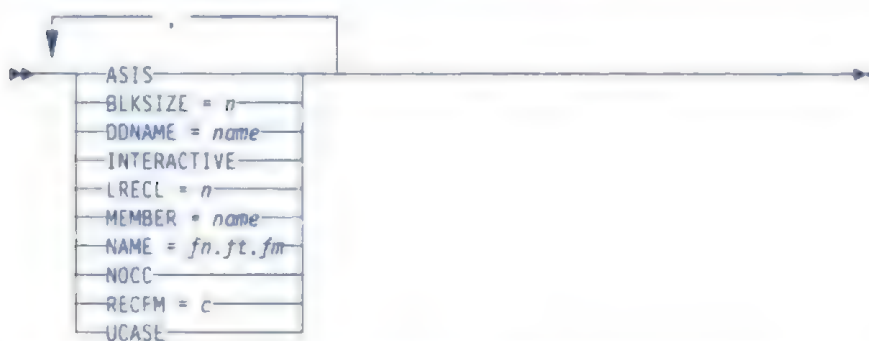


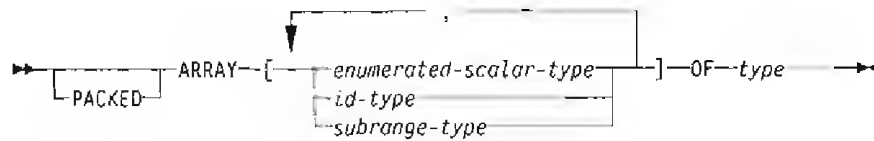
Figure 372. Syntax of Options for Opening Files

See *VS Pascal Application Programming Guide* for more information concerning these options for opening files.

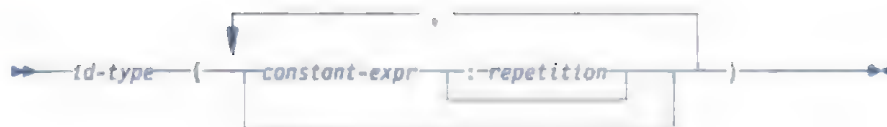
Appendix D. Syntax Diagrams

Appendix D. Syntax Diagrams

ARRAY Data Type



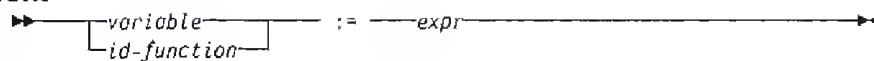
Array Structure



ASSERT Statement



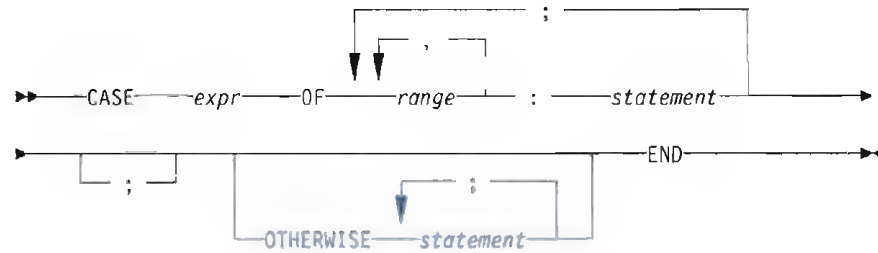
Assignment Statement



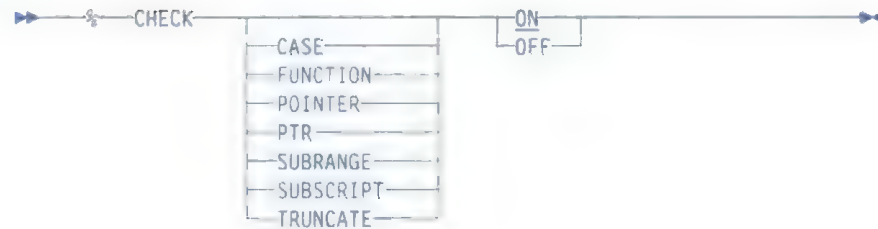
Canonical Mixed Strings



CASE Statement



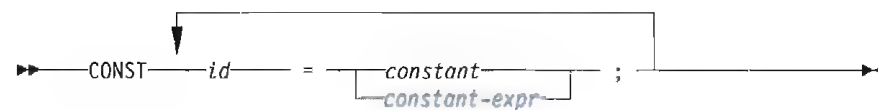
%CHECK Directive



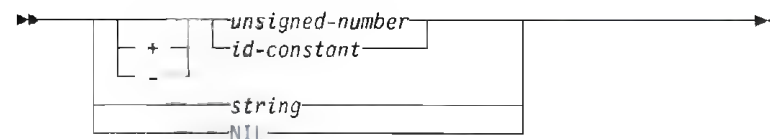
Compound Statement



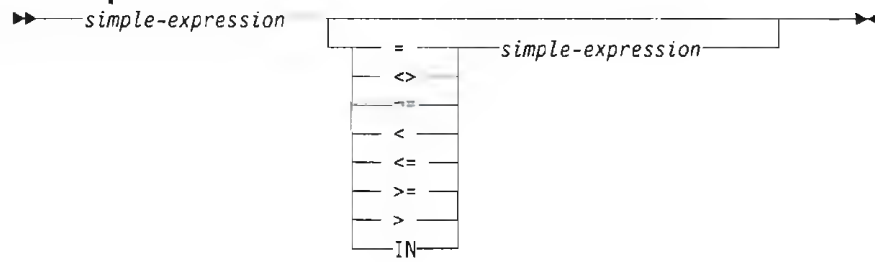
CONST Declaration



Constants



Constant Expression or Expression



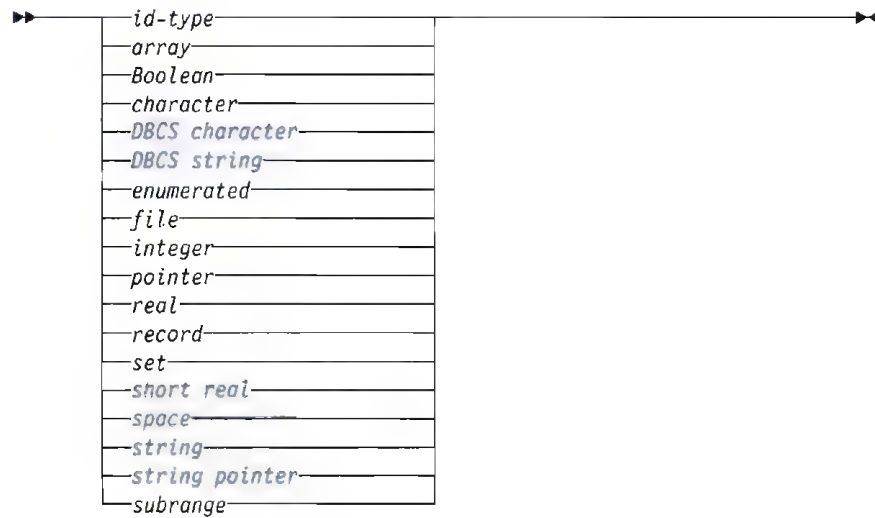
CONTINUE Statement



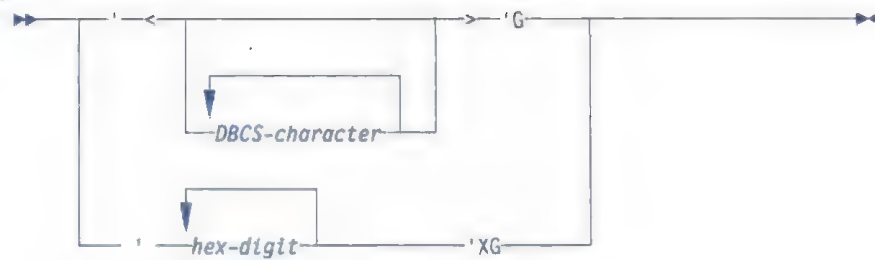
%CPAGE Directive



Data Type



DBCS String Literal



DEF Declaration



Empty Statement



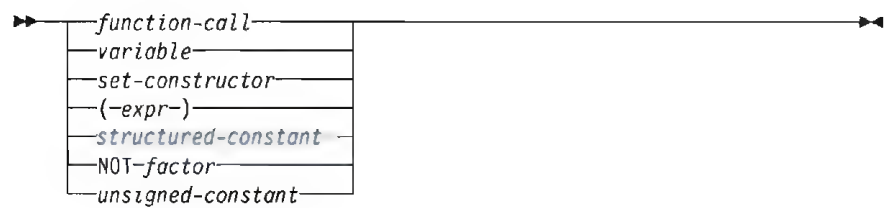
%ENDSELECT Directive



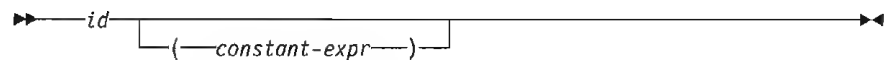
Enumerated Scalar Data Type



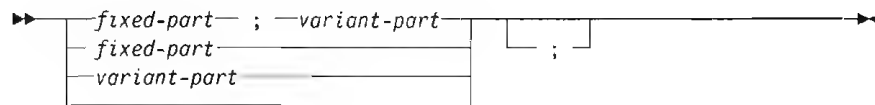
Factor



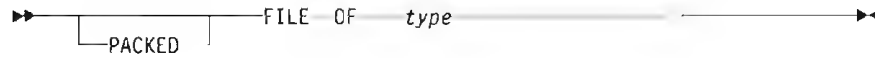
Field



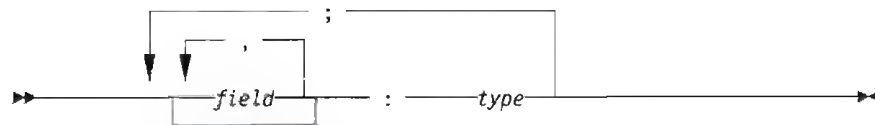
Field-list



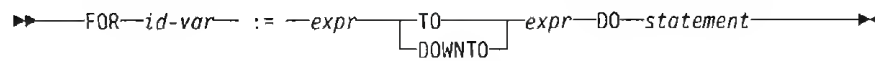
FILE Data Type



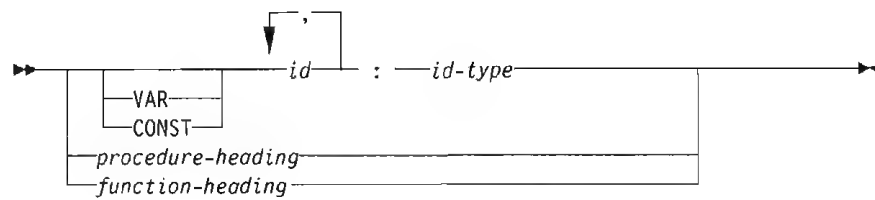
Fixed-part



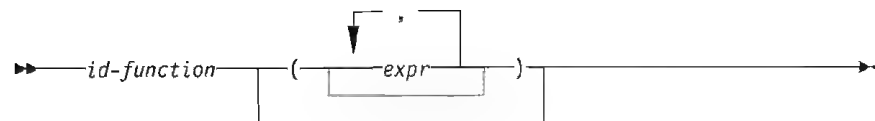
FOR Statement



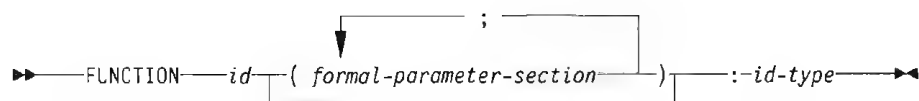
Formal Parameter Section



Function Call



Function Heading



Function-id



GOTO Statement



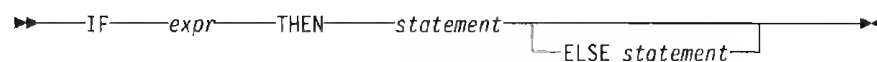
GSTRING Data Type



Identifier



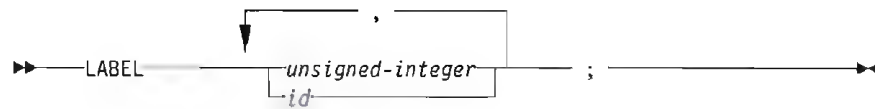
IF Statement



%INCLUDE Directive



LABEL Declaration



LEAVE Statement



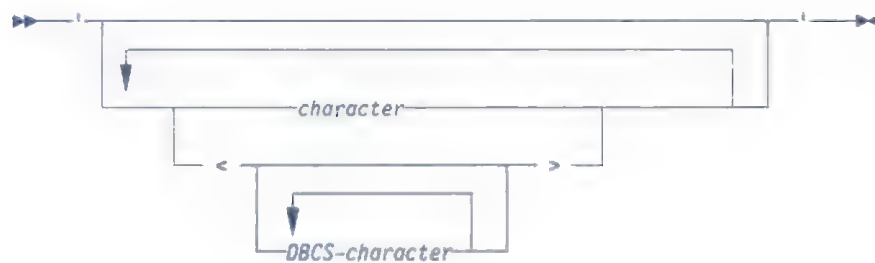
%LIST Directive



%MARGINS Directive



Mixed Strings



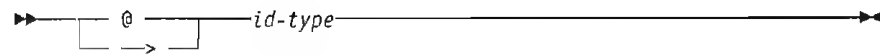
Ordinal Conversion



%PAGE Directive



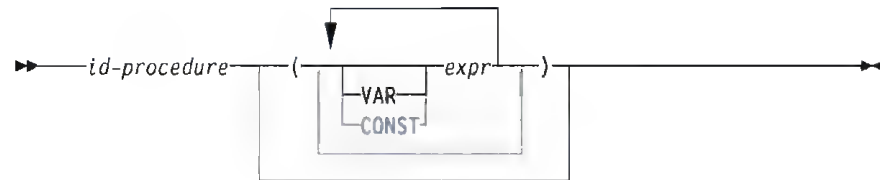
Pointer Data Type



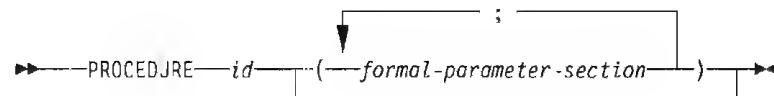
%PRINT Directive



Procedure Call



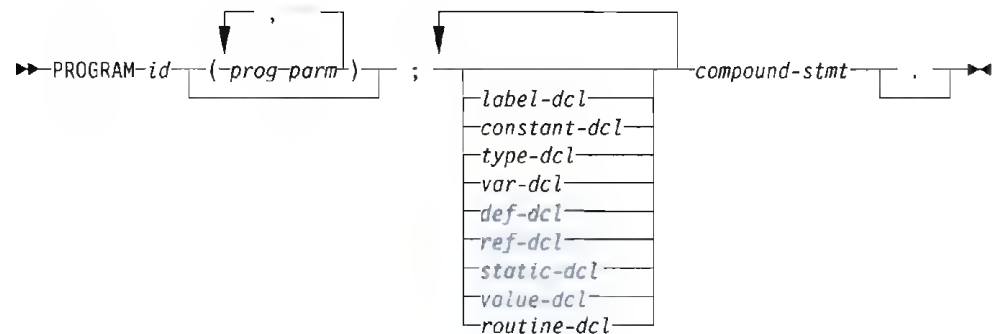
Procedure Heading



Procedure-id



Program Unit



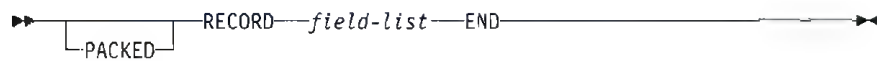
Range



Record Structure



Record Data Type



REF Declaration



REPEAT Statement



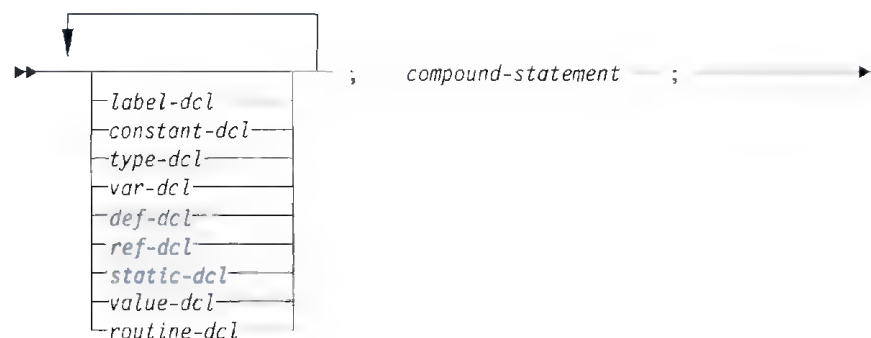
Repetition



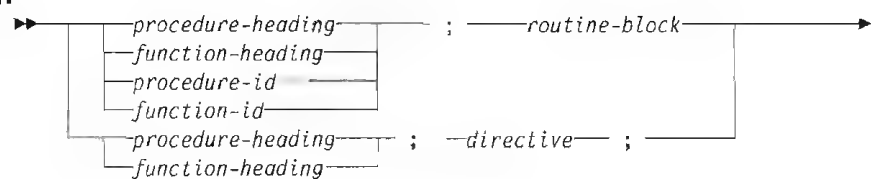
RETURN Statement



Routine Block



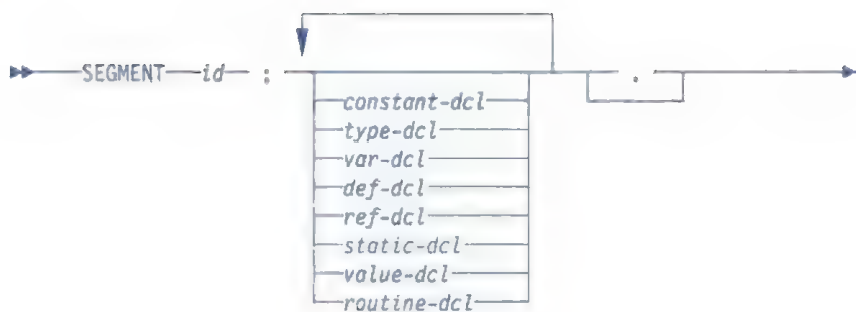
Routine Declaration



Routine Directive



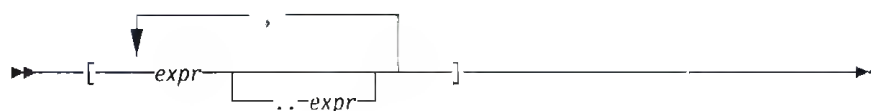
Segment Unit



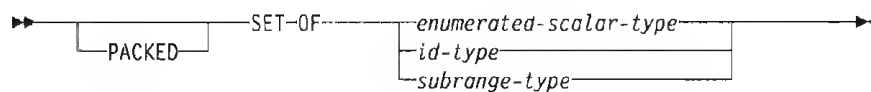
%SELECT Directive



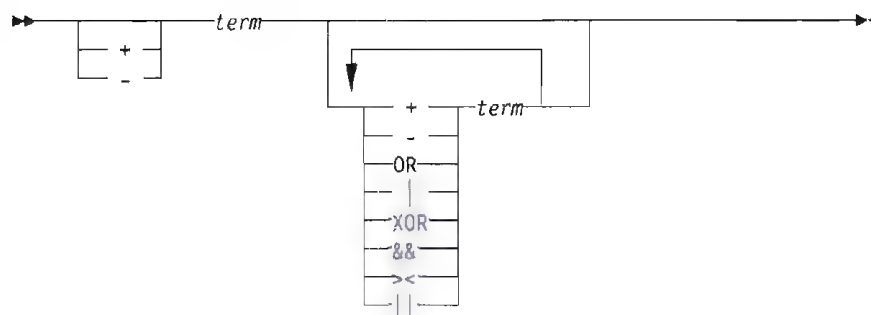
Set Constructor



SET Data Type



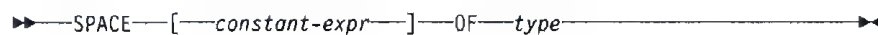
Simple Expression



%SKIP Directive



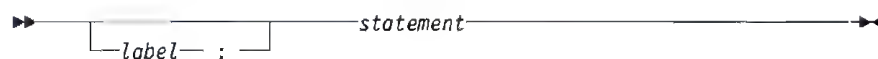
SPACE Data Type



%SPACE Directive



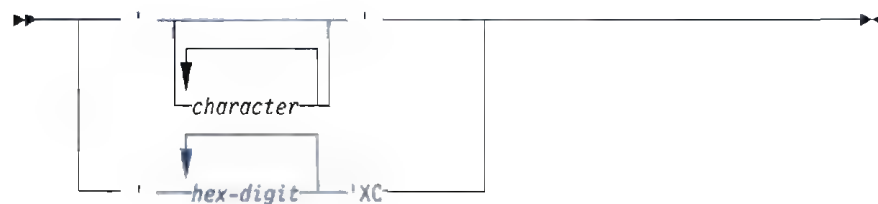
Statements



STATIC Declaration



String Literal



STRING Data Type



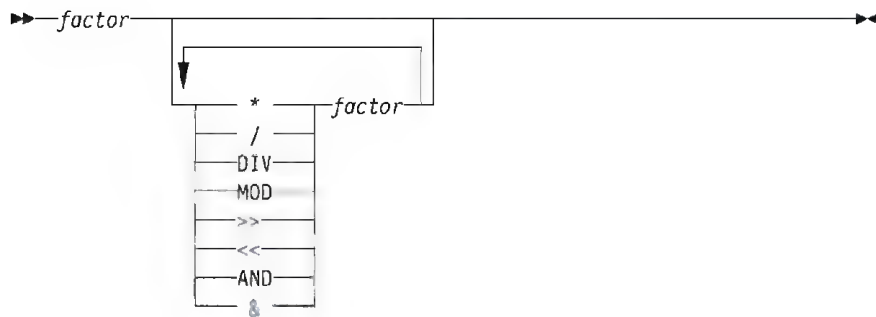
Structured Constant



Subrange Scalar Data Type



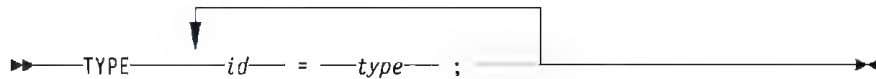
Term



%TITLE Directive



TYPE Declaration



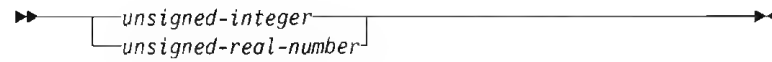
%UHEADER Directive



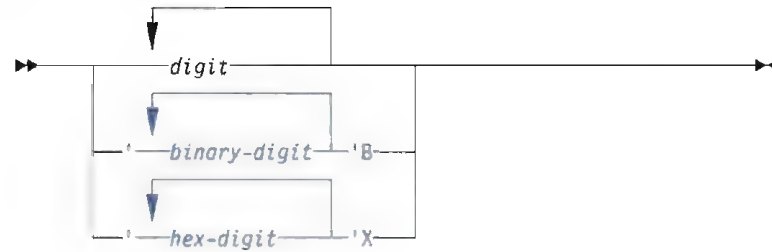
Unsigned Constant



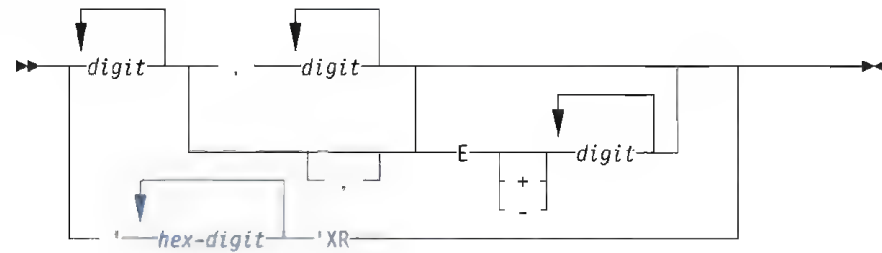
Unsigned Number



Unsigned Integer Literal



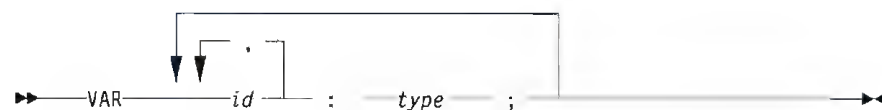
Unsigned Real Number Literal



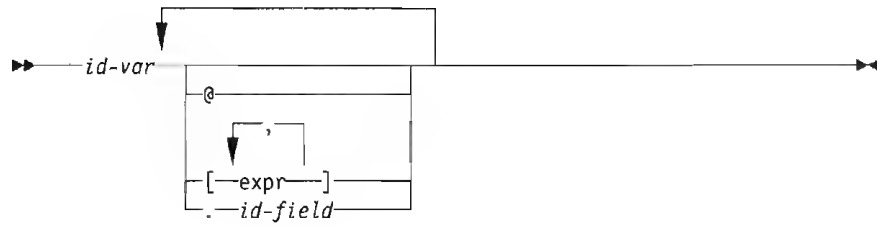
VALUE Declaration



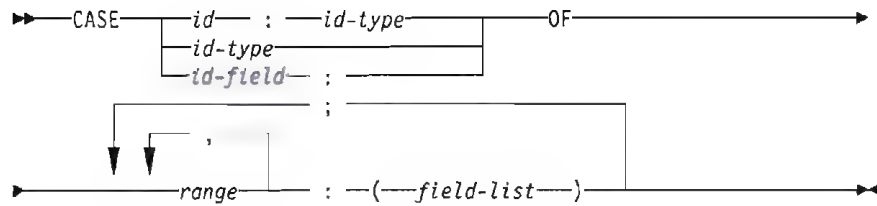
VAR Declaration



Variable Reference



Variant-part



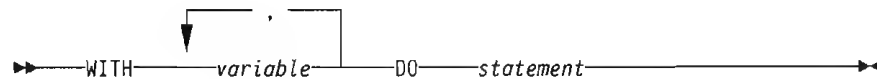
%WHEN Directive



WHILE Statement



WITH Statement



%WRITE Directive



Appendix E. Migration Considerations

Appendix E. Migration Considerations

From VS Pascal Release 1 to VS Pascal Release 2

VS Pascal Release 2 supports all existing Release 1 functions with full upward compatibility at both the source and object levels, with the exceptions noted in Figure 373.

Release 2 Change to ...	Message Issued	Nature of Change
Debugged Object Decks	—	Release 1 object decks compiled with DEBUG must be recompiled for use with Release 2. This allows the debugger to work with larger programs.
Compile-Time Limits	AMPX902S	<p>In Release 2, routines compiled with the HEADER compile-time option, which places header information in the code space, might exceed 8K. To correct the error:</p> <ul style="list-style-type: none">• Compile routines with the NOHEADER compile-time option• Issue a %UHEADER OFF before the routine if a user header is being used.
Compiler Option Checking	Messages in the Range 7xx	Release 2 now flags some invalid compiler options that were ignored in Release 1. This affects only those users who invoke the compiler with their own commands.
Compiler Directive Checking	—	Release 2 now flags some invalid compiler directives that were ignored in Release 1.
Hex String Checking	819	Release 2 flags hex string literals containing an odd number of hex digits (for example, '404'XC) as errors. Release 1 added a zero to the right of hex string literals that contained an odd number of hex digits.

Figure 373 (Part 1 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
DBCS Checking	—	<p>The GRAPHIC compile-time option now causes VS Pascal to check double-byte character set (DBCS) literals and comments and the %TITLE and %WRITE compiler directives to ensure that:</p> <ul style="list-style-type: none"> • A shift-out ('OE'X) character and shift-in ('OF'X) character are paired before the end of a source record. (This is the only check done by Release 1.) • Every shift-in is preceded by a shift-out. • There are an even number of bytes between a shift-out and shift-in. • Only valid DBCS characters occur between a shift-out and shift-in.
Structured Constant Checking	—	As defined in the proposed ANSI/IEEE Extended Pascal Standard, Release 2 now flags as errors structured constants containing files.
VAR Parameter Checking	—	Release 2 now flags as errors VAR parameters that do not have the same size.
Writing Character Data	—	As defined in the proposed ANSI/IEEE Extended Pascal Standard, when a character variable or constant is written with a field width of zero, no data will be written.
"="	—	<p>Compile-time and run-time options no longer accept "=". In Release 2 you must specify optname = optvalue as optname(optvalue).</p> <p>Note: Only the ERRCOUNT, ERRFILE, STACK, and HEAP run-time options, and the LINECOUNT and PAGEWIDTH compile-time options, are affected.</p>
Operator	—	

Figure 373 (Part 2 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
New "> <" Operator	—	<p>When the characters "> <" are passed in a string to TOKEN or LTOKEN, they are now returned as one token rather than two, unless the ">" was returned as part of another token.</p> <p>Note: The characters "> <" can now be used as a set operator for symmetric difference as defined in the proposed ANSI/IEEE Extended Pascal Standard. As an IBM extension to the Standard, these characters can also be used as Boolean exclusive or. Release 2 still supports XOR and "&&" so that existing code need not be updated. However, new code should use only "> <".</p>
MAIN and REENTRANT Routine Directives	—	Release 2 now flags as errors MAIN and REENTRANT routines that do not have their bodies declared.
ONERROR Routines	AMPX600 AMPX601 AMPX602 AMPX700 AMPX701 AMPX702 AMPX081 AMPX082 AMPX083 AMPX084	As part of National Language Support, ONERROR routines that caused message AMPX047 in Release 1 now cause messages AMPX600-602 in Release 2. Routines that caused message AMPX057 in Release 1 now cause messages AMPX700-702 in Release 2. Routines that caused message AMPX089 now cause message AMPX081 in Release 2. Routines that caused messages AMPX086 through AMPX088 in Release 1 now cause messages AMPX082-084 in Release 2.
LPAD and RPAD Procedures	—	<p>You should delete any %INCLUDE STRING directives from your source code if you want to use LPAD and RPAD with DBCS data.</p> <p>Note: If you have declared LPAD or RPAD in a higher scope than the one in which the %INCLUDE STRING was deleted, you must use different names in order to be able to access the predefined LPAD and RPAD routines.</p>
READSTR Procedure	—	When a fieldwidth of READSTR is equal to zero, the length of the string will be used as the fieldwidth; if the fieldwidth is less than zero, the absolute value of the fieldwidth will be used as the fieldwidth. This makes READSTR consistent with READ.

Figure 373 (Part 3 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

Release 2 Change to ...	Message Issued	Nature of Change
PROLOG Macro in Assembler Routines	—	<p>Any code using the PROLOG macro should be re-assembled. This allows assembler routine errors to generate an error trace-back report rather than error message AMPX902S, and improves error checking.</p> <p>The new parameter FPARMS should be added to the PROLOG macro of any assembler code that contains local variables as well as formal parameters. This helps the PROLOG macro generate code compatible with compiler-generated code, and might prevent certain memory errors.</p>

Figure 373 (Part 4 of 4). Exceptions in VS Pascal Release 2 Support of Release 1

From Pascal/VS Release 2.2 to VS Pascal Release 1

VS Pascal Release 2 supports all functions that are in Pascal/VS Release 2.2 (the Program Offering that preceded the VS Pascal Release 1 Licensed Program) with full upward compatibility at the source level except for some minor enhancements. Recompilation of the Pascal/VS source code under VS Pascal Release 2 is required.

The main differences between Pascal/VS Release 2.2 and VS Pascal Release 1 are listed in the following section; the main differences between VS Pascal Release 1 and VS Pascal Release 2 are listed in the preceding section.

Changes in	Change
Source Language Statements	<p>Threatened FOR loop indexes are now always flagged with a warning.</p> <p>Value parameters may no longer be control variables for FOR loops.</p>

Figure 374 (Part 1 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes in	Change
Compiler Options	<p>The WARNING NOWARNING compiler options are no longer supported. They have been replaced by the FLAG compiler option.</p> <p>VS Pascal supports only two language level compiler options: ANSI83 and EXTENDED. Pascal/VS supported three compiler options: STANDARD, STDRES, and EXTENDED. When LANGLVL(ANSI83) is specified, all standard violations will be flagged as compiler errors. If warnings (instead of errors) are desired, STDFLAG(W) must be specified. This is equivalent to LANGLVL(STDRES).</p> <p>You cannot specify only the option, such as EXTENDED for LANGLVL. You must specify the complete option, such as LANGLVL(EXTENDED).</p>
Run-Time Library Routines	<p>The component types of the arrays passed to the PACK and UNPACK routines must have equal ranges if they are subrange data types.</p> <p>Range checking will now be done on the integer-to-character conversion function (CHR).</p>
Function Declarations	<p>Function results must now be identifier types. This will only prevent subrange specifications whose first value is a constant identifier.</p> <p>Each function must now contain an assignment to the function result.</p>

Figure 374 (Part 2 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes in	Change
I/O Routines	<p>Output of some real numbers will be changed to conform to the 1983 ANSI/IEEE standard. Pascal/VS always writes a real number with a value of zero as "0.0" or "0." The ANSI/IEEE standard makes no such distinction between zero and other real numbers. In VS Pascal, real zero is now handled just like any other real number. If the Pascal/VS behavior is required, a statement such as:</p> <pre>IF R=0.0 THEN WRITELN (R : LENGTH1 : 1)</pre> <p>should produce results equivalent to those in Pascal/VS.</p> <p>Parameters for the I/O routines READ, READLN, READSTR, WRITE, WRITELN, and WRITESTR are now evaluated in left-to-right order.</p> <p>When writing REAL data: in the case where <i>length1</i> and <i>length2</i> are specified and <i>length2</i> equals zero, then a decimal point is written, but no decimal place is written. If <i>length2</i> is negative, the number is written using the floating point form. If the Pascal/VS Release 2.2 behavior is required, a statement such as:</p> <pre>IF (LENGTH2 = 0) AND (ABS(R) >= 1) THEN WRITELN(ROUND(R) : LENGTH1) ELSE WRITELN(R : LENGTH1 : LENGTH2);</pre> <p>will replace WRITELN(R : LENGTH1 : LENGTH2).</p> <p>The CLOSE procedure no longer accepts open options.</p>
Program Parameters	<p>Duplicate program parameters are now flagged.</p> <p>Program parameters (other than INPUT or OUTPUT) that are not declared as global variables are now flagged.</p> <p>If INPUT is specified as a program parameter, a RESET(INPUT) will be issued. If this behavior is not desired, remove INPUT from the program parameter list.</p> <p>If OUTPUT is specified as a program parameter, a REWRITE(OUTPUT) will be issued. If this behavior is not desired, remove OUTPUT from the program parameter list.</p>

Figure 3/4 (Part 3 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Changes in	Change
Operating Systems	The OS/VS 1 and VM/PC operating systems are not supported. Although VS Pascal and its generated programs may work on these systems, any problems on these systems must be reproduced on a supported operating system. The IBM Support Center will not accept problem reports using these operating systems.
Other Items	<p>Illegal use of file variables (embedded files) not flagged by Pascal/VS will now be diagnosed by VS Pascal. There were situations in which file variables occurred illegally, and Pascal/VS did not flag such occurrences as illegal. Illegal occurrences of file variables include: files within files, assigning files, files in value parameters, and files in function results.</p> <p>Variables preceded by a unary plus and variables in parentheses are no longer allowed as actual VAR parameters.</p> <p>Range checking is now done on all pass-by-value and pass-by-constant actual expression parameters.</p> <p>Fields in records may no longer have the same name as the domain type of a new pointer type being referenced in the record.</p> <p>Invalid values for value assignments are now flagged as errors.</p> <p>Global labels in segment units are now flagged with a warning message because they can't be branched to.</p> <p>All tag constants in a variant record must be legal values for the tag type of the record.</p>

Figure 3/4 (Part 4 of 4). Exceptions to VS Pascal Release 1 Support of Pascal/VS Release 2.2

Glossary

actual parameter. The actual value passed to a routine. See formal parameter.

anonymous type. A type, specified on a variable declaration, that does not use a type name.

array type. The structured type that consists of an indexed list of elements, of the same data type.

assignment compatible. The term used to indicate whether a value may be assigned to a variable.

automatic variable. A variable allocated on entry to a routine and deallocated on the subsequent return. An automatic variable is declared with the VAR declaration.

base scalar type. The data type on which another type is based.

bit. One binary digit.

byte. The unit of addressability on the System/370, its length is 8 bits.

canonical mixed string. A string in which adjacent shift-in/shift-out character pairs and DBCS nulls have been removed.

case label. A value or range of values that comes before a statement in a case statement branch. When the selector evaluates to the value of a case label, the statement following the case label is executed.

compatible types. The term that is used to indicate that operations between values of those types are permitted.

compilable unit. An independently compilable piece of code. There are two types of unit: the program unit and the segment unit.

component. The name of a value in a structured type.

constant. A value that is either a literal or an identifier that has been associated with a value in a CONST declaration.

constant expression. An expression that can be completely evaluated by the compiler at compile time.

current heap. The area of storage in the VS Pascal run-time environment where dynamic variables allocated by calls to NEW will reside. While many heaps can exist at one time, there is only one current heap.

DBCS. See double-byte character set.

dereference. Accessing a dynamic variable pointed to by a pointer.

double-byte character set (DBCS). A set of characters where each character requires 2 bytes. Languages such as Kanji require such double-byte representations.

double-byte character set (DBCS) fixed string. A PACKED ARRAY [1..n] OF GCHAR.

dynamic variable. A variable that is allocated under programmer control. Explicit allocations and deallocations are required; the predefined procedures NEW and DISPOSE are provided for this purpose.

EBCDIC. See extended binary-coded decimal interchange code.

element. The component of an array.

entry routine. A procedure or function that can be invoked from outside the unit in which it is defined. The routine is called an entry point in the program in which it is defined. An entry routine cannot be imbedded in another routine; it must be declared in the outermost level of a unit.

enumerated scalar type. A scalar that is defined by enumerating the elements of the type. Each element is represented by an identifier.

executable program. Consists of object code from your main program that is link-edited with the object code from the run-time library and any segments needed by the main program.

extended binary-coded decimal interchange code (EBCDIC). The underlying character set used in VS Pascal.

external routine. A procedure or function whose body is not contained in the unit being compiled.

external variable. A variable that can be referenced from other units and scopes than the one in which it was declared.

field. The component of a record.

file pointer. May be thought of as a pointer into an input/output buffer.

FILE type. A data type that is the mechanism to do input and output in VS Pascal.

fixed part (of a record). That part of a record that exists in all instances of a particular record type.

fixed string. See single-byte character set (SBCS) fixed string and double-byte character set (DBCS) fixed string.

floating-point number. A subset of the set of real numbers

formal parameter. A parameter as declared in the routine heading. A formal parameter is used to specify what is permitted to be passed to a routine. See "actual parameter"

function. A routine that is invoked by coding its name in an expression. The routine passes a result back to the invoker through the routine name.

GENERIC routine. A routine whose parameter count, types, and passing mechanism are specified by the call to the routine

heap. An area of storage where dynamic variables are created.

hexadecimal digits. A digit that is a member of the set of sixteen digits: 0 through 9, and then A through F used in a number system of Base 16.

hexadecimal graphic data. Hexadecimal digits used in DBCS data. The number of hexadecimal digits must be a multiple of four.

identifier. The name of a declared item.

index. The selection mechanism applied to an array to identify an element of the array.

integer. The set of positive and negative whole numbers.

internal routine. A routine that can be used only from within the lexical scope in which it was declared.

invoker. That piece of code that calls another function, procedure, or unit.

lexical scope. Identifies the portion of a unit in which a name is known. An identifier declared in a routine is known within that routine and within all nested routines. If a nested routine declares an item with the same name, the outer item is not available in the nested routine.

mixed string. A string consisting of a mixture of DBCS and SBCS characters.

offset. The selection mechanism of a space. An element is selected by placing an integer value in brackets. The origin of a space is based on zero.

operand. The data that is being manipulated by an operator.

operator. A mathematical or logical process that is used to manipulate data.

ordinal type. A scalar type whose values are mapped to a continuous range of integers

partitioned data set. A file (sometimes referred to as a library) containing logical files that are called members.

pass-by-CONST. The parameter passing mechanism by which an expression, variable, or constant is passed to the called routine. The called routine is not permitted to modify the formal parameter. If the actual parameter is an expression, a temporary parameter will be created and its address will be passed to the called routine.

pass-by-read-only-reference. See "pass-by-CONST."

pass-by-read/write-reference. See "pass-by-VAR."

pass-by-value. The parameter passing mechanism by which a copy of the value of the actual parameter is passed to the called routine. If the called routine modifies the formal parameter, the corresponding actual parameter is not affected.

pass-by-VAR. The parameter passing mechanism by which the address of a variable is passed to the called routine. If the called routine modifies the formal parameter, the corresponding actual parameter is changed. Only variables may be passed by this means. Fields of a packed record or elements of a packed array may not be passed as VAR parameters in standard Pascal.

pointer. A variable that contains the address of a dynamic variable.

procedure. A routine, invoked by coding its name as a statement, that does not pass a result back to the invoker.

program. See executable program.

program unit. The name of the compilable unit of code that represents the first unit executed.

record type. The structured data type that contains a series of fields. Each field can be a different data type. A field is selected by the name of the field.

reserved word. An identifier whose use is restricted by the VS Pascal compiler.

routine. A unit of a VS Pascal program that may be called. The two types of routines are functions and procedures.

scalar type. A type whose values contain only one element.

SBCS. See single-byte character set.

segment unit. A compilable unit in VS Pascal that is used to contain entry routines.

selector. The term in a CASE statement that, once evaluated, determines which of the possible branches the CASE statement will execute.

SET type. Used to define a variable that represents all combinations of elements of some ordinal type.

shift-in character. Indicates the end of DBCS data and is denoted by X'0F'.

shift-out character. Indicates the beginning of DBCS data and is denoted by X'0E'.

single-byte character set (SBCS). A set of characters where each character requires one byte.

single-byte character set (SBCS) fixed string. A PACKED ARRAY [1..N] OF CHAR.

SPACE type. Used to define a variable whose components may be positioned at any byte in the total space of the variable.

statement. The executable code in a VS Pascal program.

string. Represents an ordered list of characters whose size may vary at run time. There is a maximum size for every string.

string constant. A string whose value is fixed by the compiler.

structured type. Any one of several data type mechanisms that defines variables that have multiple values. Each value is referred to generally as a component.

subrange type. Used to define a variable whose value is restricted to some subset of values of a base ordinal type.

subheap. An area in a heap delimited by a call to MARK. Subheaps are treated in a stack-like manner within a heap.

tag field. The field of a record that defines the structure of the variant part.

type. Defines the permissible values a variable may assume.

type definition. The specification of a data type. The specification may appear in a type declaration or in the declaration of a variable.

type identifier. The name given to a declared type.

unit. See compilable unit.

variant part (of a record). That portion of a record that may vary from one instance of the record to another. The variant portion consists of a series of variants that may share the same physical storage.

Bibliography

VS Pascal Publications

These books provide additional information about VS Pascal.

Evaluation

- *VS Pascal General Information*, GC26-4318, provides an overview of VS Pascal.
- *VS Pascal Licensed Program Specifications*, GC26-4317, contains warranty information for VS Pascal.

Application Programming

- *VS Pascal Application Programming Guide*, SC26-4319, explains how to compile, execute, and debug VS Pascal programs.
- *VS Pascal Reference Summary*, SX26-3760, provides quick-reference charts of VS Pascal language rules and processing/debugging options.

Installation

- *VS Pascal Installation and Customization for VM*, SC26-4342, explains how to install VS Pascal under VM/SP and VM/XA.
- *VS Pascal Installation and Customization for MVS*, SC26-4321, explains how to install VS Pascal under MVS/SP, MVS/XA and MVS/ESA.

Diagnosis

- *VS Pascal Diagnosis Guide and Reference*, LY27-9525, explains how to diagnose, report, and request information on VS Pascal-related problems

Related Publications

Many text books are available on Pascal. The following list does not reflect any preference and is not exhaustive, it is merely provided to start the reader in the right direction. You may wish to check your library for other books or more recent editions.

- *Standard Pascal User Reference Manual* by Doug Cooper, W. W. Norton & Company, Inc., 1983, 176 pages.
- *Oh! Pascal!* by Doug Cooper and Michael Clancy, W. W. Norton & Company, Inc., 1985, 607 pages.
- *PASCAL: An Introduction to Methodical Programming* by W. Findlay and D. Watt, Computer Science Press, 1978, 306 pages; UK Edition by Pitman International Text, 1978.
- *Pascal for the 80s* by Samuel Grier, Brooks/Cole Publishing Company, 1985, 540 pages.
- *Programming in PASCAL* by Peter Grogono, Addison-Wesley, Reading Mass., 1978.
- *Pascal Users Manual and Report* by K. Jensen and N. Wirth, Springer-Verlag, New York, 1978.
- *Structured Programming and Problem-Solving with Pascal* by R.B. Kieburtz, Prentice-Hall Inc., 1978.
- *Programming via Pascal* by J.S. Rohl and Barrett, Cambridge University Press.
- *An Introduction to Programming and Problem-Solving with Pascal* by G.M. Schneider, S.W. Weingart and D.M. Periman, Wiley & Sons Inc., New York, 394 pages.
- *The Pascal Handbook*, by Jacques Tiberghien, SYBEX Inc., 1981.
- *Introduction to Pascal* by J. Weisn and J. Elder, Prentice-Hall Inc., Englewood Cliffs, 220 pages.
- *A Practical Introduction to Pascal* by I.P. Wilson and A.M. Addyman, Springer-Verlag New York, 1978, 145 pages; MacMillan, London, 1978.
- *Paradigms and Programming with Pascal* by Derrick Wood, Computer Science Press, Inc., 1984.

Index

Special Characters

- < operator 199
- << operator 197
- <> operator 199
- <= operator 199
- + operator 198
- | operator 198
- || operator 198
- && operator 198
- * operator 197
- ¬ operator 197, 199
- operator 198
- / operator 197
- % directives
 - see compiler directives
- > operator 199
- >< operator 198
- >> operator 197
- >= operator 199
- = operator 199

A

- ABS function 118
- actual parameters 105
- addition operators 198
- additional routines
 - CMS procedure 186
 - ITOHs function 187
 - LPAD procedure 187
 - ONERROR procedure 188
 - PICTURE function 189
 - RPAD procedure 191
- ADDR function 118
- ALFA data type 50
 - applicable operators and predefined functions 50
- ALFALEN constant 37
- ALPHA data type 50
 - applicable operators and predefined functions 50
- ALPHALEN constant 37
- AND operator 197
- ARCTAN function 118
- array constants 38
- ARRAY data type 51
 - applicable predefined functions 52
- arrays
 - multidimensional 51, 93
 - packed 51
 - referencing 93
 - subscripting 93
- ASSERT statement 209
- assignment compatibility 48

- assignment statement 209
 - restrictions on usage 210
- automatic variables 31

B

- bas c data type 44
- BOOLEAN data type 53
 - applicable operators and predefined functions 53
 - applicable relational operators 54
- BOOLEAN expressions 199
 - order of evaluation 199
 - short circuiting 200
- boundary alignments of variables 49

C

- canonical mixed strings 83, 117
- case labels 212
- case sensitivity
 - identifiers 8
 - literals 14
 - reserved words 9
- CASE statement 211
 - labels 212
 - selectors 211
- CHAR data type 55
 - applicable operators and predefined functions 55
- CHECK compiler directive 231
- CHR function 119
- CLOCK function 119
- CLOSE procedure 119
- CMS procedure 186
- COLS function 119
- comments 11
 - double-byte character set (DBCS) 12
 - nested 12
 - restrictions with %WHEN 12
 - restrictions with MVS batch 12
- compatible data types 47
- compilable unit 18
- compiler directives 230
 - %CHECK 231
 - %CPAGE 232
 - %ENDSELECT 233
 - %INCLUDE 233
 - %LIST 234
 - %MARGINS 235
 - %PAGE 235
 - %PRINT 235
 - %SELECT 236
 - %SKIP 236
 - %SPACE 236
 - %TITLE 237

compiler directives (*continued*)

- %UHEADER 237
- %WHEN 238
- %WRITE 240
- list and summary 230
- used for conditional compilation 239
- compound statement 214
- COMPRESS function 120
- conditional compilation
 - using %SELECT, %WHEN, and %ENDSELECT 239
- conformant string parameters 105, 106
- CONST declaration 27
- constant expressions 201
 - applicable predefined functions 201
 - relation to VS Pascal and Standard Pascal 36
- constants 36
 - categories
 - NIL 36
 - strings 36
 - TRUE and FALSE 36
 - unsigned integers 36
 - predefined
 - list and summary 37
 - relation to VS Pascal and Standard Pascal 36
 - structured 37
- CONTINUE statement 215
- conversion routines
 - CHR function 119
 - FLOAT function 126
 - GSTR function 126
 - GTOSTR function 127
 - list and summary 113
 - ORD function 150
 - ROUND function 165
 - STOGSTR function 168
 - STR function 169
 - TRUNC function 174
- COS function 121
- CPAGE compiler directive 232

D

- data inquiry routines
 - ADDR function 118
 - HBOUND function 128
 - HIGHEST function 129
 - LBOUND function 130
 - list and summary 114
 - LOWEST function 132
 - MAX function 137
 - MIN function 139
 - ODD function 149
 - PRED function 152
 - SIZEOF function 167
 - SUCC function 171
- data movement routines
 - list and summary 114
 - PACK procedure 150

data movement routines (*continued*)

- UNPACK procedure 175
- data type compatibility 46
 - and empty set 47
 - and NIL 47
 - assignment compatibility 48
 - compatible types 47
 - implicit conversions 46
 - same types 47
- data types
 - ALFA 50
 - ALPHA 50
 - ARRAY 51
 - basic 44
 - BOOLEAN 53
 - CHAR 55
 - DBCS fixed string 56
 - enumerated 57
 - FILE 59
 - GCHAR 60
 - GSTRING 61
 - INTEGER 64
 - list and summary 49
 - pointer 44, 66
 - REAL 67
 - RECORD 69
 - SBCS fixed string 76
 - SET 77
 - SHORTREAL 79
 - simple 44
 - SPACE 81
 - STRING 81
 - STRINGPTR 45, 84
 - structured 45
 - subrange 86
 - TEXT 88
 - user-defined 45
- DATETIME procedure 121
- DBCS (double-byte character set) comments 12
- DBCS fixed string data type 56
 - applicable operators and predefined routines 56
- declarations 24
 - CONST 27
 - DEF 28
 - LABEL 28
 - list and summary 24
 - order of declaration 27
 - REF 29
 - STATIC 30
 - TYPE 30
 - VALUE 31
 - VAR 31
- DEF declaration 28
 - DEF variables 28
 - used in VALUE declarations 31
- DELETE function 122
- directives
 - see routine directives

directives, compiler
 see compiler directives
DISPOSE procedure 123
DISPOSEHEAP procedure 123
DIV operator 66, 197
double-byte character set (DBCS) comments 12
dynamic variables 66, 96

E

empty set 47
empty statement 215
ENDSELECT compiler directive 233
enumerated data type 57
 applicable predefined functions 58
EOF function 124
EOLN function 125
EPSREAL constant 37, 67
EXP function 125
expressions 194
 BOOLEAN 199
 constant 201
 logical 202
 order of evaluation 194
EXTERNAL routine directive 109
external variables 28

F

FALSE BOOLEAN constant 37
field referencing 95
fields 69
 naming 70
 offset qualification 75
FILE data type 59
 applicable functions 60
 restrictions on usage 59
file pointers 59
file referencing 97
fixed string data types
 DBCS 56
 SBCS 76
FLOAT function 126
FOR statement 216
 restrictions on usage 217
 used with DOWNT0 216
 used with TO 216
formal parameters 105
formal routine parameters 106
FORTRAN routine directive 109
 restrictions on usage 109
FORWARD routine directive 110
function calls 203
function results 107
functions 102
 ABS 118
 ADDR 118
 ARCTAN 118
 CHR 119

functions (*continued*)

CLOCK 119
COLS 119
COMPRESS 120
COS 121
DELETE 122
EOF 124
EOLN 125
EXP 125
FLOAT 126
GSTR 126
GTOSTR 127
HBOUND 128
HIGHEST 129
INDEX 130
IT0HS 187
LBOUND 130
LENGTH 131
LN 132
LOWEST 132
LTRIM 135
MAX 137
MAXLENGTH 137
MCOMPRESS 138
MDELETE 139
MIN 139
MINDEX 140
MLENGTH 140
MLTRIM 141
MRINDEX 141
MSUBSTR 142
MTRIM 143
ODD 149
ORD 150
ordinal conversion 203
PARMS 151
PICTURE 189
PRED 152
RANDOM 154
RINDEX 164
ROUND 165
SIN 167
SIZEOF 167
SQR 167
SQRT 168
STOGSTR 168
STR 169
SUBSTR 170
SUCC 171
TRIM 174
TRUNC 174

G

- GCHAR data type 60
 - applicable operators and predefined functions 61
- general routines
 - HALT procedure 128
 - list and summary 114
 - TRACE procedure 173
- GENERIC routine directive 110
 - communication with other products 110
 - restrictions on usage 112
- GET procedure 126
 - restrictions on usage 126
- global automatic variables 32
- global identifiers 24
- GOTO statement 219
 - restrictions on usage 219
- GSTR function 126
- GSTRING data type 61
 - applicable operators and predefined functions 62
 - binary operators applied 63
 - indexing 95
- GSTRING subscripting 95
- GTOSTR function 127

H

- HALT procedure 128
- HBOUND function 128
- heap 66
- HIGHEST function 129

I

- I/O routines
 - CLOSE procedure 119
 - COLS function 119
 - EOF function 124
 - EOLN function 125
 - GET procedure 126
 - list and summary 114
 - PAGE procedure 151
 - PDSIN procedure 151
 - PDSOUT procedure 152
 - PUT procedure 153
 - READ procedure (record files) 154
 - READ procedure (text files) 155
 - READLN procedure (text files) 155
 - RESET procedure 163
 - REWRITE procedure 163
 - SEEK procedure 166
 - TERMIN procedure 171
 - TERMOUT procedure 171
 - UPDATE procedure 176
 - WRITE procedure (record files) 177
 - WRITE procedure (text files) 178
 - WRITELN procedure (text files) 178
- identifiers 8
 - case sensitivity 8

identifiers (*continued*)

- global 24
- lexical scope 24
- local 24
 - restrictions on format 8
- IF statement 220
 - nesting 221
- implicit type conversions 46
- IN operator 199
- INCLUDE compiler directive 233
- INDEX function 130
- INPUT file 88
 - usage requirements 19
- INTEGER data type 64
 - applicable operators and predefined functions 64
 - DIV and MOD operators 66
 - MAXINT and MININT constants 64
 - ranges of integers 86
 - storage mapping 64
- ITOH function 187

L

- LABEL declaration 28
 - in statements 208
 - used with GOTO statement 219
- LBOUND function 130
- LEAVE statement 222
- LENGTH function 131
- lexical scope
 - of identifiers 24
 - of nested routines 25
- linking units to form a program 22
- LIST compiler directive 234
 - used with small sections of units 234
- literals 12
 - case sensitivity 14
 - double-byte character set (DBCS) hexadecimal 15
 - double-byte character set (DBCS) mixed 15
 - floating-point hexadecimal 15
 - integer binary 15
 - integer hexadecimal 14
 - string hexadecimal 15
- LN function 132
- local identifiers 24
- logical expressions 202
 - and INTEGERS 202
 - applicable logical operators 202
- LOWEST function 132
- LPAD procedure 133, 187
- LTOKEN procedure 134
- LTRIM function 135

M

MAIN routine directive 112
 communication with other languages 112
 restrictions on usage 112
MARGINS compiler directive 235
MARK procedure 136
mathematical routines
 ABS function 118
 ARCTAN function 118
 COS function 121
 EXP function 125
 list and summary 115
 LN function 132
 RANDOM function 154
 SIN function 167
 SQR function 167
 SQRT function 168
MAX function 137
MAXCHAR constant 37
MAXINT constant 37, 64
MAXLENGTH function 137
MAXREAL constant 37, 67
MCOMPRESS function 138
MDELETE function 139
migration considerations 273
MIN function 139
MINDEX function 140
MININT constant 37, 64
MINREAL constant 37, 67
MLENGTH function 140
MLTRIM function 141
MOD operator 66, 197
MRINDEX function 141
MSLBSTR function 142
MTRIM function 143
multidimensional arrays 51, 93
multiplication operators 197
mutually recursive routines 110

N

nesting of programs 25
NEW procedure 143
 used with pointer data type 66
 used with STRINGPTR data type 84
NEWHEAP procedure 146
NIL 47
NOT operator 197

O

ODD function 149
ONERROR procedure 188
operators 197
 addition 198
 and operands 197
 four classes by precedence 194
 multiplication 197

operators (*continued*)

 NOT 197
 relational 199
OR operator 198
ORD function 150, 203
ordinal conversion functions 203
OUTPUT file 88
 usage requirements 19

P

PACK procedure 150
PACKED ARRAY OF CHAR data type 81
packed arrays 51
packed records 74
PAGE compiler directive 235
PAGE procedure 151
parameters
 actual 105
 conformant string 105, 106
 formal 105
 formal routine 106
 non-predefined routines as parameters 107
 pass by read-only reference 106
 pass-by-CONST 106
 pass-by-reference 105
 pass-by-value 105
 pass-by-VAR 105
 predefined routines as parameters 107
 restrictions on routines as parameters 107
parameters, program
 see program parameters 19
PARMS function 151
Pascal/VS, migration considerations 273
pass by read-only reference parameters 106
pass-by-CONST parameters 106
pass-by-reference parameters 105
pass-by-value parameters 105
pass-by-VAR parameters 105
PDSIN procedure 151
PDSOUT procedure 152
PICTURE function 189
pointer data type 44, 66
 applicable operators and predefined routines 67
 declaring 66
 used with NEW procedure 66
pointer referencing 96
 checking errors 97
pointer variables 66
PRED function 152
predefined constants 37
predefined variables 93
PRINT compiler directive 235
procedure call 223
procedures 102
 CLOSE 119
 CMS 186
 DATETIME 121

procedures (*continued*)

- DISPOSE 123
- DISPOSEHEAP 123
- GET 126
- HALT 128
- LPAD 133, 187
- LTOKEN 134
- MARK 136
- NEW 143
- NEWHEAP 146
- ONERROR 188
- PACK 150
- PAGE 151
- PDSIN 151
- PDSOUT 152
- PUT 153
- QUERYHEAP 153
- READ (record files) 154
- READ (text files) 155
- READLN (text files) 155
- READSTR 160
- RELEASE 162
- RESET 163
- RETCODE 163
- REWRITE 163
- RPAD 165, 191
- SEEK 166
- TERMIN 171
- TERMOUT 171
- TOKEN 172
- TRACE 173
- UNPACK 175
- UPDATE 176
- USEHEAP 177
- WRITE (record files) 177
- WRITE (text files) 178
- WRITELN (text files) 178
- WRITESTR 184

program elements

- comments 11
- identifiers 8
- literals 12
- reserved words 9
- special symbols 10

program parameters

- for Standard Pascal 19
- for VS Pascal 20

program unit 18

- structure 18

PUT procedure 153

- restrictions on usage 153

Q

QUERYHEAP procedure 153

R

RANDOM function 154

READ procedure (record files) 154

READ procedure (text files) 155

- reading
 - CHAR data 158
 - DBCS fixed string data 158
 - GCHAR data 158
 - GSTRING data 158
 - integer data 159
 - real and shortreal data 159
 - SBCS fixed string data 159
 - string and mixed string data 160
 - variables with a length 157

reading

- CHAR data 158
- DBCS fixed string data 158
- GCHAR data 158
- GSTRING data 158
- integer data 159
- real and shortreal data 159
- record file data 154
- SBCS fixed string data 159
- string and mixed string data 160
- text file data 155
- variables with a length 157

READLN procedure (text files) 155

- reading
 - CHAR data 158
 - DBCS fixed string data 158
 - GCHAR data 158
 - GSTRING data 158
 - integer data 159
 - real and shortreal data 159
 - SBCS fixed string data 159
 - string and mixed string data 160
 - variables with a length 157

READSTR procedure 160

REAL data type 67

- and EPSREAL constant 67
- and MAXREAL constant 67
- and MINREAL constant 67
- applicable operators and predefined functions 68

record constants 39

RECORD data type 69

- applicable predefined functions 76
- declaring 70

record files 59

records 69

- fields 69
 - naming 70
 - offset qualification 75
- fixed part 71
- packed 74

- records (*continued*)
 - storage mapping 74
 - tag field 71
 - variant part 71
 - variant selector 71
- recursive function 107
- REENTRANT routine directive 112
 - restrictions on usage 113
- REF declaration 29
 - REF variables 29
 - restrictions on usage 29
- referencing
 - field 95
 - file 97
 - pointer 96
 - space 98
- relational operators 199
 - and DBCS and mixed strings 199
- RELEASE procedure 162
- REPEAT statement 223
- reserved words 9
 - case sensitivity 9
 - list and summary 9
 - special usage 9
- RESET procedure 163
- RETCODE procedure 163
 - restrictions on usage 163
- RETURN statement 224
- REWRITE procedure 163
- RINDEX function 164
- ROUND function 165
- routine declarations 102, 105
 - parameters
 - see parameters
- routine directives 108
 - EXTERNAL 109
 - FORTRAN 109
 - FORWARD 110
 - GENERIC 110
 - MAIN 112
 - REENTRANT 112
- routines 102
 - returning data 102
 - see also
 - additional routines
 - conversion routines
 - data inquiry routines
 - data movement routines
 - general routines
 - I/O routines
 - mathematical routines
 - storage management routines
 - string routines
 - system access routines
- RPAD procedure 165, 191

S

- same data types 47
- SBCS fixed string data type 76
 - applicable operators and predefined routines 76
- SEEK procedure 166
- segment unit 20
- SELECT compiler directive 236
 - restrictions on usage 236
- selectors 211
- set constructors 204
- SET data type 77
 - applicable operators and functions 79
 - packed set 78
 - storage mapping 78
- short circuiting 200
- SHORTREAL data type 79
 - and floating-point data 79
 - applicable operators and predefined functions 80
- simple data type 44
- SIN function 167
- SIZEOF function 167
- SKIP compiler directive 236
- SPACE compiler directive 236
- SPACE data type 81
 - applicable functions 81
- space referencing 98
 - checking errors 99
- special symbols 10
 - list and summary 10
 - special usage 11
- SQR function 167
- SQRT function 168
- statements 208
 - ASSERT 209
 - assignment 209
 - CASE 211
 - compound 214
 - CONTINUE 215
 - empty 215
 - FOR 216
 - GOTO 219
 - IF 220
 - LEAVE 222
 - list and summary 208
 - procedure call 223
 - REPEAT 223
 - RETURN 224
 - WHILE 225
 - WITH 225
- STATIC declaration 30
 - static variables 30
 - restriction on usage 30
 - used in VALUE declarations 31
- STOGSTR function 168
- storage allocation of variables 49
- storage management routines
 - DISPOSE procedure 123
 - DISPOSEHEAP procedure 123

- storage management routines (*continued*)
 - list and summary 116
 - MARK procedure 136
 - NEW procedure 143
 - NEWHEAP procedure 146
 - QUERYHEAP procedure 153
 - RELEASE procedure 162
 - USEHEAP procedure 177
- STR function 169
- STRING data type 81
 - and PACKED ARRAY OF CHAR 81
 - and subscripted string arrays 94
 - applicable byte-oriented operators and routines 82
 - applicable character-oriented operators and routines 83
 - binary operators applied 83
 - compatibility 82
 - used with DBCS data 82
 - used with LENGTH function 82
 - used with MAXLENGTH function 82
 - used with mixed data 82
 - used with SBCS data 82
- string routines
 - COMPRESS function 120
 - DELETE function 122
 - INDEX function 130
 - LENGTH function 131
 - list and summary
 - for mixed strings 117
 - for SBCS and DBCS strings 116
 - LPAD procedure 133
 - LTOKEN procedure 134
 - LTRIM function 135
 - MAXLENGTH function 137
 - MCOMPRESS function 138
 - MDELETE function 139
 - MINDEX function 140
 - MLENGTH function 140
 - MLTRIM function 141
 - MRINDEX function 141
 - MSUBSTR function 142
 - MTRIM function 143
 - READSTR procedure 160
 - RINDEX function 164
 - RPAD procedure 165
 - SUBSTR function 170
 - TOKEN procedure 172
 - TRIM function 174
 - WRITESTR procedure 184
- string subscripting 94
- STRINGPTR data type 45, 84
 - applicable operators and predefined routines 85
 - used with NEW procedure 84
- structured constants 37
 - array 38
 - record 39
- structured data type 45
- subrange data type 86
 - applicable predefined functions 87
 - packed 86
 - ranges of integers 86
 - restrictions on usage 87
- subscripting
 - arrays 93
 - errors, checking 95
 - GSTRING variables 95
 - string variables 94
- SUBSTR function 170
- SUCC function 171
- syntax diagrams
 - how to read
 - default parameters 5
 - multiple parameters 4
 - no parameters 2
 - optional parameters 3
 - required parameters 2
- system access routines
 - CLOCK function 119
 - DATETIME procedure 121
 - list and summary 117
 - PARMS function 151
 - RETCODE procedure 163

T

- tag field 71
- TERMIN procedure 171
- TERMOUT procedure 171
- TEXT data type 88
 - applicable routines 88
 - INPUT and OUTPUT file 88
- TITLE compiler directive 237
- TOKEN procedure 172
- TRACE procedure 173
- TRIM function 174
- TRUE BOOLEAN constant 37
- TRUNC function 174
- type compatibility 46
- TYPE declaration 30

U

- UHEADER compiler directive 237
- units
 - linking to form a program 22
 - program 18
 - segment 20
- UNPACK procedure 175
- UPDATE procedure 176
- USEHEAP procedure 177

V

- VALUE declaration 31
 - and STATIC and DEF variables 31
- VAR declaration 31
 - errors in usage 32
- variables 92
 - automatic 31
 - boundary alignments 49
 - DEF variables
 - restrictions on usage 28
 - when to specify 32
 - external 28
 - global automatic 32
 - static 30
 - storage allocation 49
- variant part of a record 71
- variant selector of a record 71
- VS Pascal
 - migration considerations 273

W

- WHEN compiler directive 238
 - notes on usage 239
- WHILE statement 225
- WITH statement 225
- WRITE compiler directive 240
- WRITE procedure (record files) 177
- WRITE procedure (text files) 178
 - writing
 - Boolean data 180
 - CHAR data 181
 - DBCS fixed string data 181
 - expressions with a length 180
 - GCHAR data 181
 - GSTRING data 182
 - integer data 182
 - real and shortreal data 182
 - SBCS fixed string data 183
 - string and mixed string data 183
- WRITELN procedure (text files) 178
 - writing
 - Boolean data 180
 - CHAR data 181
 - DBCS fixed string data 181
 - expressions with a length 180
 - GCHAR data 181
 - GSTRING data 182
 - integer data 182
 - real and shortreal data 182
 - SBCS fixed string data 183
 - string and mixed string data 183
- WRITESTR procedure 184
 - writing
 - Boolean data 180
 - CHAR data 181
 - DBCS fixed string data 181
 - expressions with a length 180

writing (*continued*)

- GCHAR data 181
- GSTRING data 182
- integer data 182
- real and shortreal data 182
- record file data 177
- SBCS fixed string data 183
- string and mixed string data 183
- text file data 178

X

- XOR operator 198

SC26-4320-1

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Comments (please include specific chapter and page references) :

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information:

Name _____ Date _____

Company _____ Phone No. (_____) _____

Address _____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page)

Reader's Comment Form

Fold and tape

Please do not staple

Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023



Fold and tape

Please do not staple

Fold and tape



SC26-4320-1

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____

Comments (please include specific chapter and page references) :

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information:

Name _____ Date _____

Company _____ Phone No. (_____) _____

Address _____

Thank you for your cooperation. No postage is necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail them directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape

Please do not staple

Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Programming Publishing
P.O. Box 49023
San Jose, CA 95161-9023



Fold and tape

Please do not staple

Fold and tape





Program Number
5668-767
5668-717

File Number
S370-40

The VS Pascal Library

Application Programming Guide	SC26-4319
Diagnosis Guide and Reference	LY27-9525
General Information	GC26-4318
Installation and Customization for MVS	SC26-4321
Installation and Customization for VM	SC26-4342
Language Reference	SC26-4320

Supplementary Publications

Licensed Program Specifications	GC26-4317
Reference Summary	SX26-3760

SC26-4320-1

